Class Notes for I547/N547

Christopher Raphael

April 6, 2010

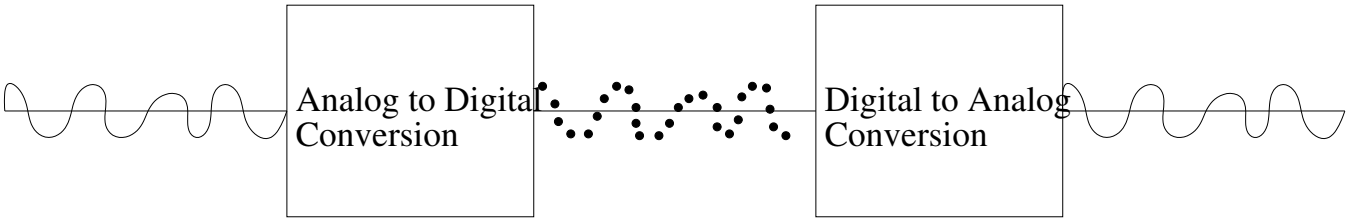
Chapter 1

Preliminaries

1.1 Sampled Audio

Sound is created by moving objects, such as a tuning fork that vibrates back and forth after it is struck. The motion of the tuning fork creates changes in air pressure around the tuning fork. This time-varying air pressure propagates through the air in all directions. If this time-varying pressure reaches some other object (like your eardrum) it will cause that object to move in the same way as the tuning fork originally did — only a little later in time due to the time it takes the "signal" to propagate through the medium (usually air for us). Thus we can think of sound as either time-varying pressure, or as time-varying displacement (movement).

Sampled audio represents sound as a sequence of numbers that are evenly spaced measurements of pressure (or displacement).



The two most relevant parameters of the sampling process are the sample rate and bit depth:

sample rate The *sampling rate* (SR) is the number of samples per unit time. This is usually measured in Hz = cycles (samples) per second, or kHz = 1000s of samples per second. For instance

1. Audio on a compact disc is sampled at 44.1 kHz.
2. Audio on DAT = digital audio tape is often sampled at 48 kHz.
3. Internet telephony usually has sample rates around 8 kHz.

As the sample rate decreases, the sound quality degrades. While the converse of this is also true, there are arguments that say that, for humans, there is no point in having sampling rates much larger than 40 kHz.

bit depth Samples are usually represented on a computer as integers (fixed point) as opposed to floating point. The *bit depth* is the number of bits alloted to each sample. The most common sample representation is as *signed* integers. For instance, if our bit depth is 4 then our sample values go from

$$\begin{array}{rcl} 1000 & = & -8 \\ 1001 & = & -7 \\ \vdots & \vdots & \vdots \\ 1111 & = & -1 \\ 0000 & = & 0 \\ 0001 & = & 1 \\ \vdots & \vdots & \vdots \\ 0111 & = & 7 \end{array}$$

The reason we interpret 1111 as -1 for a signed integer is that when we add 0001 = 1 to 1111 = -1 (ignoring the carry) we get 0. Similarly for the other negative signed integers.

Typical bit depth = 16. There are some representation schemes such as "mulaw" that use non-evenly-spaced samples, though we won't use this in our treatment. Sound quality degrades as the bit depth increases.

Examples

The "Audio quality" experiments on the class web page,

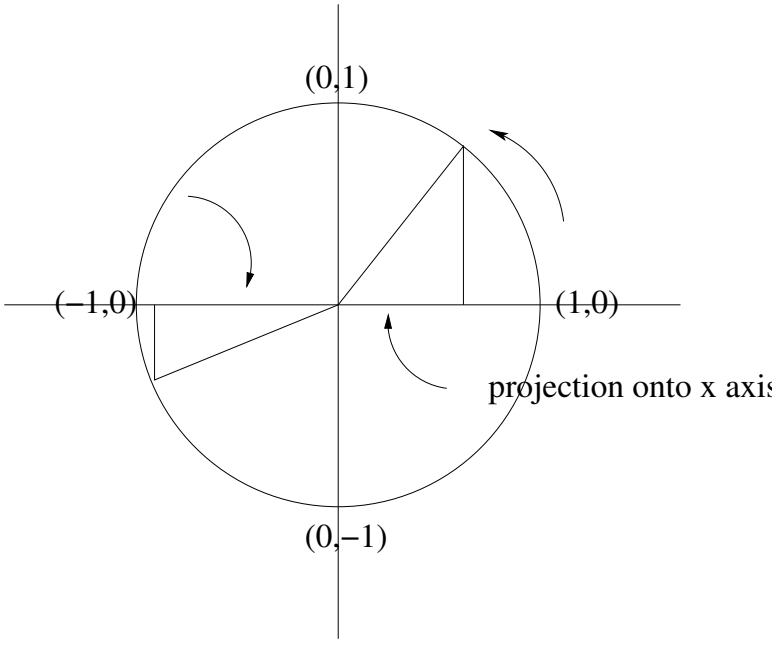
http://www.music.informatics.indiana.edu/courses/I547

show the opening of the 2nd movement of the Mozart oboe concerto with different bit depths and sampling rates.

1.2 The Sine Wave

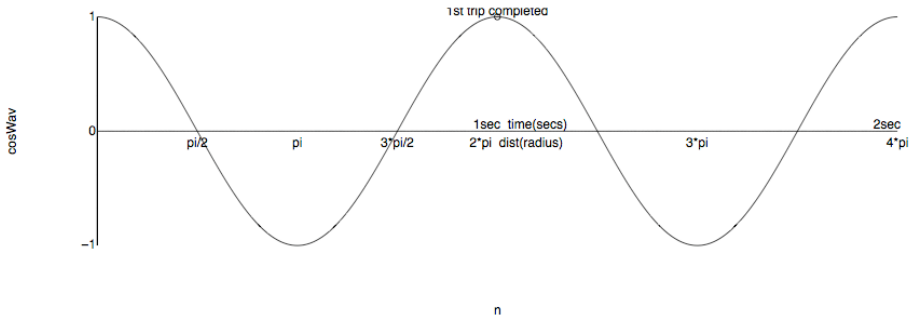
The *unit circle* is the circle with radius 1 centered at the "origin" (0,0).

Imagine we start at the point (1,0) and move counterclockwise around the circle making one full cycle each second (that is, we move at at rate of 1Hz). At every time we observe the projection of our position onto the x-axis

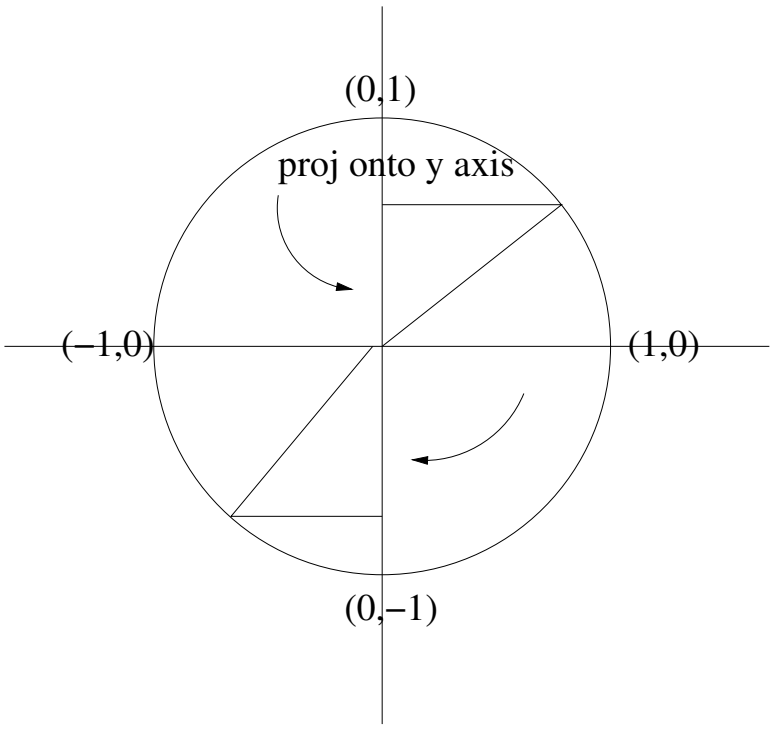


We could think of the projection as a function of the arc length traveled around the circle, rather than time. This projection, as a function of the arc length traveled is the *cosine* function $\cos(t)$.

$\cos(t)$ = signed projection onto x axis after traveling arc length of t around unit circle

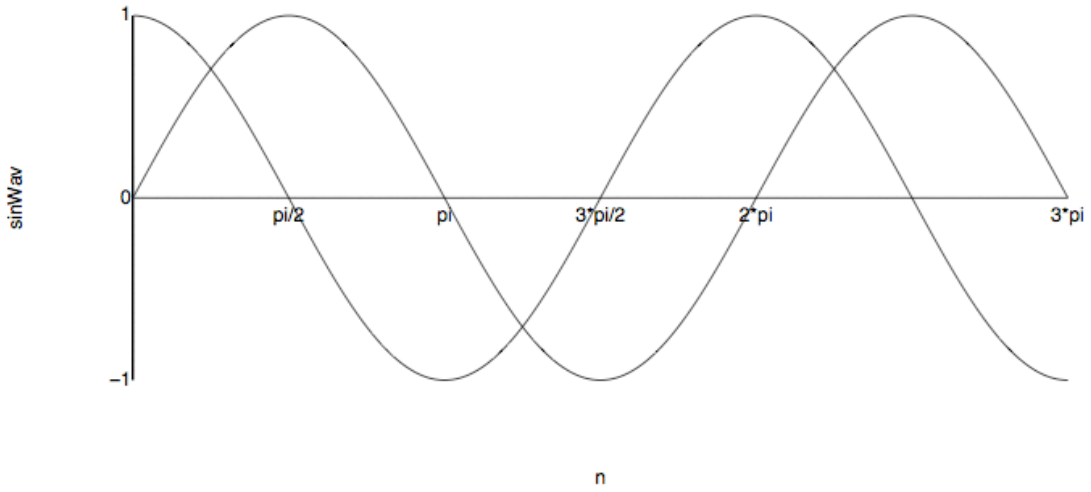


If we imagine the projection onto the y axis, analogously, we get the *sine* function.

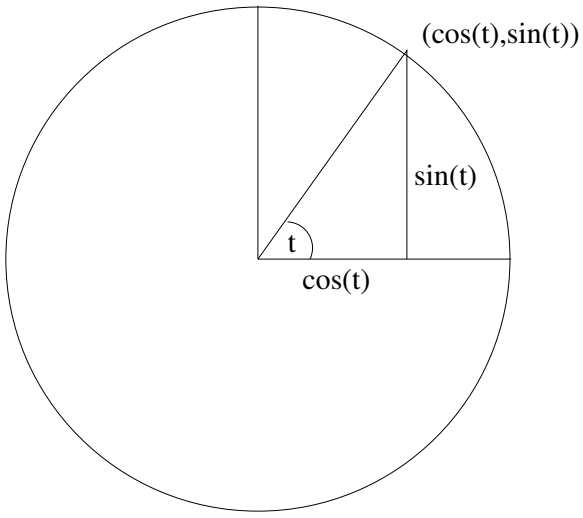


$\sin(t)$ = signed projection onto y axis after traveling arc length of t around unit circle

Note: since the projection onto the x axis starting at (1,0) ($t = 0$), is the same as the projection onto the y axis starting at (0,1) $t = \pi/2$. we have $\cos(t) = \sin(t + \pi/2)$. That is, the sine and cosine functions are the same except for a shift in time, as shown below:



This discussion can be summarized in a single picture showing that the point on the unit circle making an angle of t with the x axis has coordinates $(x = \cos(t), y = \sin(t))$.



1.2.1 What Does Sine Sound Like? (simple_sine.r)

A sine wave is a "pure" tone.

Frequency

$\sin(t)$ complete its cycle (period) every 2π (every trip around circle), so if t measures seconds, $\sin(2\pi t)$ oscillates once every second. Similarly, $\sin(2\pi 2t)$ oscillates twice every second and

$$\sin(2\pi f t) \text{ oscillates } f \text{ times per second.}$$

We say that the frequency of $\sin(2\pi ft)$ is f Hz $= f$ cycles per second.

Using the **simple_sine.r** R program on the web page, listen to sine wave with the the following frequencies:

$\sin(2\pi 440t)$ a sine at 440 Hz. This is the "A" that musicians often tune to.

$\sin(2\pi 450t)$ a sine at 450 Hz. a little "higher" than before.

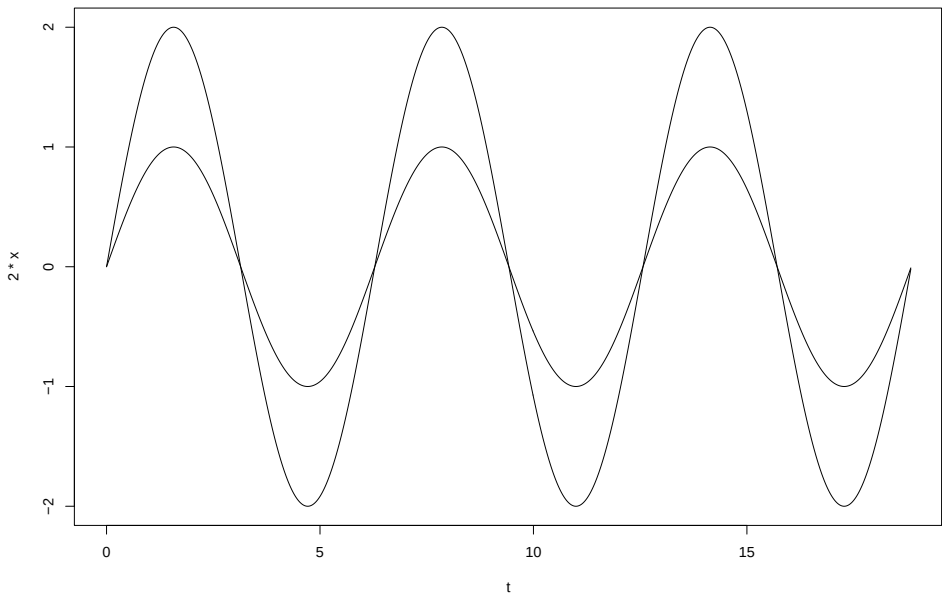
$\sin(2\pi 430t)$ a sine at 430 Hz. a little "lower than before.

$\sin(2\pi 880t)$ $880 = 2 \times 440$ Hz. an *octave* above 440 Hz.

$\sin(2\pi 1760t)$ $1760 = 2 \times 880$ Hz. 2 octaves above 440 Hz.

Amplitude

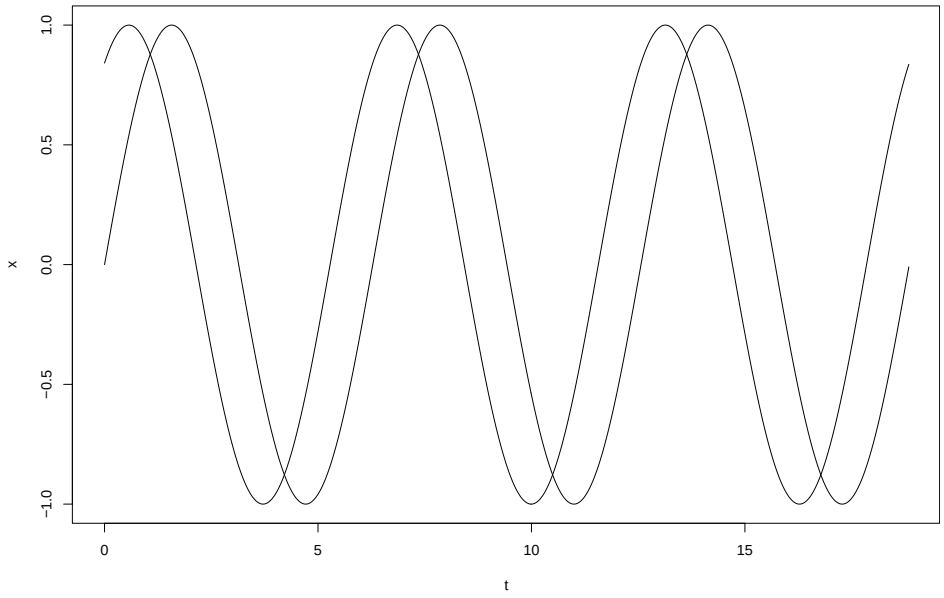
The following figure shows $\sin(2\pi ft)$ vs. $2\sin(2\pi ft)$



For positive a, $a\sin(2\pi ft)$ oscillates between $-a$ and $+a$. Construct the $a\sin(2\pi ft)$ for $a = 1, .1, .01, \ldots$ using the **simple_sine.r** program and listen to the sounds. You will hear the sounds becoming quieter and quieter.

Phase

The figure below shows $a\sin(2\pi ft)$ vs $a\sin(2\pi ft + 1)$.



For any ϕ, $a\sin(2\pi ft + \phi)$ is sine "shifted" earlier (to the left) by π radians. We will call ϕ the *phase* of the sine wave. Listen to $a\sin(2\pi ft + \phi)$ for $\phi = 0, 1$. There is no difference in the sound of the sine as we change the phase.

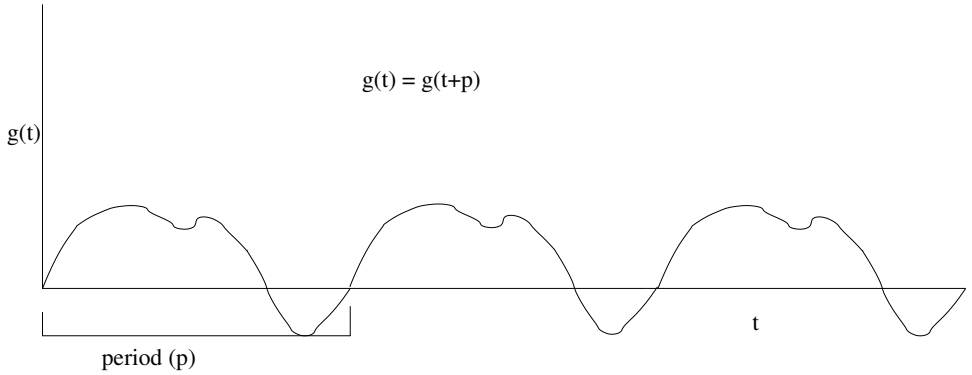
1.2.2 Periodicity

Have observed that sines at freqs. f and $2f$ sound similar (e.g. differ by octave). While we will never completely explain why this happens, we will observe some ways in which these two sine waves are similar from a mathematical perspective.

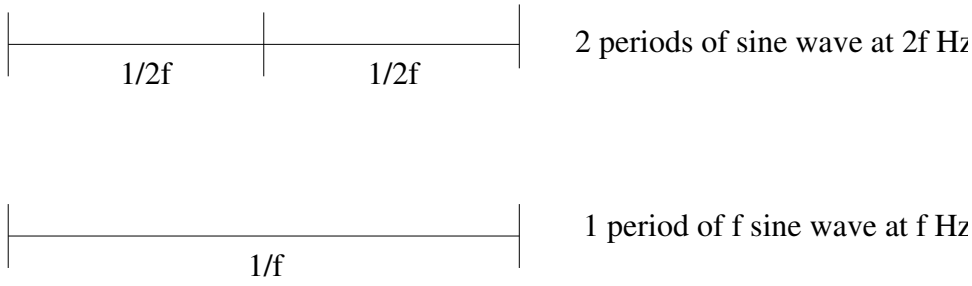
If a function repeats same shape over and over, it is said to be *periodic*. The *period* is the time for the function to repeat. That is

$$g(t) = g(t + p)$$

says that g is periodic with period p, or, more briefly, g is p-periodic. A an example of a periodic function is shown below.



For sine at f Hz ($\sin(2\pi ft)$) have f cycles/sec., so 1 cycle must take $\frac{1}{f}$ secs. That is, $\sin(2\pi ft)$ is $\frac{1}{f}$-periodic. Similarly, sine at $2f$ Hz has period of $\frac{1}{2f}$ which is half the period length of f Hz sine.



Note that, while sine at $2f$ Hz has period $\frac{1}{2f}$, it *also* has period $\frac{1}{f}$. Thus both sines at f and $2f$ are $\frac{1}{f}$-periodic. This partly explains why they sound similar to one another. We will come back to this idea later.

1.3 Musical Intervals and Frequency Ratios (intervals.r,repeated_intervals.r)

We have observed that the frequencies f and $2f$ make an "octave." Musically speaking, suppose we wanted to sing a melody that begins with an octave such as the first two notes of *Somewhere Over the Rainbow*. Regardless of what frequency we choose for the first note, say f Hz., the frequency of $2f$ Hz. will give the correct frequency for the 2nd note.

The musical term for pitch "distance" is *interval*. Thus the octave, which corresponds to the frequency ratio of 2:1 is an example of a musical interval. There are other ratios that correspond to familiar musical intervals which are tabulated below, along with melodies that begin with the interval. Use the **intervals.r** program to observe how each ratio generates an interval, and how this interval does not depend on the initial frequency.

Ratio	Interval	Example
2/1 (2:1)	Octave	*Somewhere over the Rainbow*
3/2 (3:2)	Perfect Fifth	*Also Sprach Zarathustra* (a.k.a. theme from 2001)
4/3 (4:3)	Perfect Fourth	*Here comes the Bride*
5/4 (5:4)	Major Third	*Kumbayah* (**1** 2 **3** in major)
6/5 (6:5)	Minor Third	*Chopin Funeral March* (**1** 2 **3** in minor)

Now consider the **repeated_intervals.r** program. In this program we construct a sequence of sine waves beginning with an arbitrary frequency for the first note and getting the subsequent frequencies by multiplying the previous frequency by the constant $c = 1.414$. What do we hear?

First note that every time we multiply the frequency by c the pitch moves up by the same interval. The musical name for this interval is an *augmented fourth*. By the construction of the program, the frequencies of the notes must be

$$f, cf, c^2 f, c^3 f \ldots$$

We can also hear that traversing two augmented 4ths brings us 1 octave above where we began (note that every other note is the "same.") Thus we have $c^2 = 2$, hence c $= \sqrt{2} \approx 1.414$. Thus multiplying a frequency by $\sqrt{2}$ moves the pitch up an augmented fourth.

We could also let $c = \sqrt[3]{2} = 2^{1/3}$ (the "third root" of 2). When we generate the series of pitches with frequencies $f, cf, c^2 f, c^3 f, \ldots$ we hear that each note is a Major 3rd above its predecessor (remember *Kumbayah*: $\sqrt[3]{2} \approx 5/4$). We also see that traversing this interval 3 times brings us one octave above where we began since the frequencies are $f, 2^{1/3} f, 2^{2/3} f, 2^{3/3} f = 2f, \ldots$. This follows from the rules for exponents since $(2^{1/3})^k = 2^{k \times 1/3} = 2^{k/3}$. Thus the Major 3rd splits the octave into 3 equal pieces.

If we continue this experiment dividing the octave into $4, 5, 6, \ldots$ equal pieces we get

Ratio	Interval	Example
$2^{1/2}$	Augmented 4th	*Maria* or *The Simpsons*
$2^{1/3}$	Major 3rd	*Kumbayah*
$2^{1/4}$	Minor 3rd	(**Chopin Funeral March** or **1** 2 **3** in minor scale)
$2^{1/5}$	#$%!&	ugh!
$2^{1/6}$	Major 2nd	(**1 2** in major scale)

Equal Temperament

Recall that the simple ratio for the major 3rd was 5/4. All things being equal, simple ratio intervals are often preferred by musicians as the most "correct" tuning (more on this later). However, there is also a problem with simple ratio intervals. Suppose we begin at a particular frequency and construct a series of pitches by moving up by a simple ratio major 3rd (by 5/4). In this case we have

$$f, \frac{5}{4}f, \frac{25}{16}f, \frac{125}{64}f \neq 2f$$

If f corresponds to the note C, then we would get the notes C, E, G#, C, with the problem that the 2nd C is not quite twice the frequency of the first C.

Every scheme for generating a collection of frequencies for pitches following simple ratios runs up against a problem of this kind. The most common answer to this problem is the notion of *equal temperament* (ET). The goal of equal temperament is to split the octave into 12 equal pieces. In Western music history the decision to divide the octave into 12 pieces *preceded* the notion of equal temperament. Really, ET could be applied to any number of notes in the octave, though we will see an interesting connection between 12 and ET. We have learned that, in order to get

12 equal *musical* intervals, we must have the same frequency ratio between each pair of frequencies. This is easily accomplished by the sequence:

$$f, 2^{1/12}f, 2^{2/12}f, 2^{3/12}f, \ldots, \underbrace{2^{12/12}f = 2f}_{\text{octave above } f}$$

where the initial frequency, f, is chosen arbitrarily (say A = 440 Hz).

The main advantage of ET is that every "version" of the same interval (e.g. a minor third) sounds the same. For instance, say

$$\underbrace{f}_{\text{C}}, \underbrace{2^{1/12}f}_{\text{C}\sharp}, \underbrace{2^{2/12}f}_{\text{D}}, \underbrace{2^{3/12}f}_{E\flat}, \underbrace{2^{4/12}f}_{E}, \underbrace{2^{5/12}f}_{F}, \underbrace{2^{6/12}f}_{F\sharp}, \ldots$$

Then we have

$$\begin{array}{rcl} \frac{\text{E}\flat}{\text{C}} & = & \frac{2^{3/12}f}{f} = 2^{3/12} \\ \frac{\text{E}}{\text{C}\sharp} & = & \frac{2^{4/12}f}{2^{1/12}f} = 2^{3/12} \\ \frac{\text{F}}{\text{D}} & = & \frac{2^{5/12}f}{2^{2/12}f} = 2^{3/12} \\ \vdots & \vdots & \vdots \end{array}$$

Of course, this argument holds for any interval, not just the minor third. A consequence of this tuning is that all the different keys "sound" the same, as far as the intervals are concerned.

All of this discussion is based on the number the number $2^{1/12} = \sqrt[12]{2}$. It is often useful to know the value of this constant:

$$2^{1/12} = \sqrt[12]{2} \approx 1.059$$

In other words, the frequencies of a half step (any half step) differ by about 6%. If you understand the argument leading to ET tuning, then you also see, by identical reasoning, that your money will double in 12 years if compounded at 6% annually.

Frequency vs. Midi

The Musical Instrument Digital Interface (MIDI) protocol associates a number with every possible musical pitch. The scheme assigns the numbers

$$\begin{array}{rcl} \vdots & \vdots & \vdots \\ 59 & = & \text{B below middle C} \\ 60 & = & \text{middle C} \\ 61 & = & \text{C}\sharp \text{ above middle C} \\ \vdots & \vdots & \vdots \end{array}$$

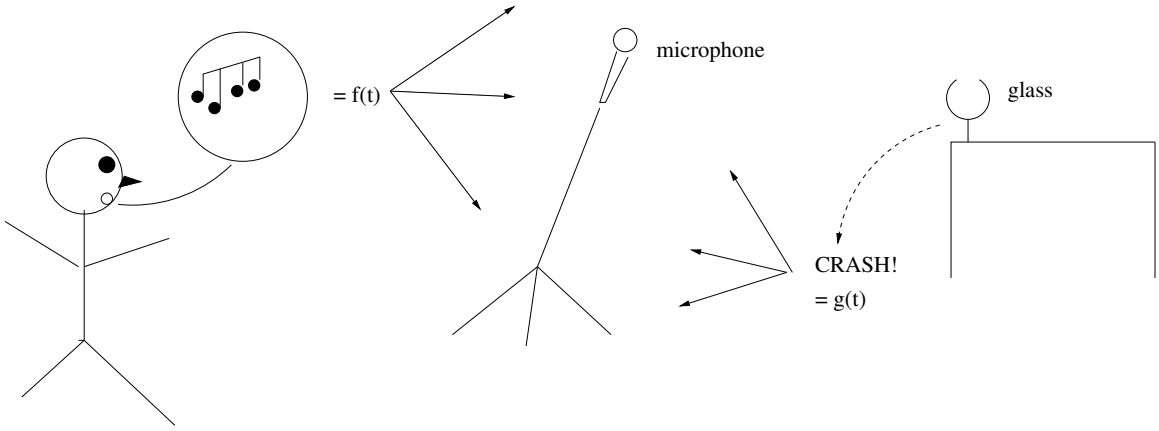
From this scheme we see that "tuning A" (A above middle C) is midi pitch 69. A common reference for this note is A = 440 Hz. If we let $f(m)$ be the frequency of the ETT midi pitch using A = 440 Hz, we get

$$f(m) = 440 \times 2^{\frac{m-69}{12}}$$

as a useful formula. From this formula we see that $f(69) = 440$ and that

$$\frac{f(m+i)}{m} = \frac{440 \times 2^{\frac{m+i-69}{12}}}{440 \times 2^{\frac{m-69}{12}}} = 2^{\frac{i}{12}}$$

Simultaneous Sounds



Consider the picture in which we have two sound sources:

1. A person singing $f(t)$
2. A glass falling of a table and breaking $g(t)$

Both sources generate sound ($f(t)$ and $g(t)$) that will propagate through the room. If we measure the result of *both* sounds at some point, say where the microphone is located, we get a signal $h(t)$. How does $h(t)$ relate to $f(t)$ and $g(t)$?

Amazing Fact #1

$$h(t) = f(t) + g(t)$$

See the class web page for a demonstration of this.

Perfect Intervals (stability_test.r)

We have observed that sine waves at frequencies f and $2f$ make an octave. We also have observed that both are $1/f$-periodic. Presumably the "simplicity" and "stability" of the octave comes from this shared periodicity. The program **stability_test.r** plays an interval followed by both notes sounding *together*. Use the program to see how intervals formed by simple ratios of integers (2:1, 3:1, 3:2, 4:3, etc.) sound stable and simple when played together. Why?

Consider sine waves at $2f$ and $3f$.

- The sine at $2f$ has a period of $\frac{1}{2f}$, but is also $\frac{1}{f}$-periodic.
- The sine at $3f$ has a period of $\frac{1}{3f}$, but is also $\frac{1}{f}$-periodic.

So both have $\frac{1}{f}$ periodicity leading to stability of the perfect fifth. Note that when the sines are summed (both pitches played together) the sum is $\frac{1}{f}$-periodic

Similarly, sines at $3f$ and $4f$ both are $\frac{1}{f}$-periodic leading to the stability of the perfect fourth. The same argument applies to other perfect intervals (though this terminology is not usual) such as 5/4 (major third) and 6/5 (minor third).

Clearly, from a musical standpoint as we look at the frequency ratio $\frac{n+1}{n}$ the interval becomes less stable. The way to understand this is to keep the lower note fixed at freq. f, so the upper note is $\frac{n+1}{n}f$. In this case

$$\begin{aligned} f &= n \times \frac{f}{n} \\ \frac{n+1}{n}f &= (n+1) \times \frac{f}{n} \end{aligned}$$

are both $\frac{n}{f}$-periodic. So as n increases the *length* of the common period gets longer. It seems that it is the *length* of this common period (the period of the sum) that relates to stability.

There is an interesting "dual" to this argument. If we keep the frequency ratio $\frac{n+1}{n}$ fixed (i.e. keep the musical interval fixed) and let f decrease, the common period length, $\frac{n}{f}$, increases. In other words. A fixed dyad (two notes sounding at once) has a longer joint period as the lower note decreases. This corresponds to the conventional musical wisdom that the same interval becomes increasing more "opaque" and "dissonant" as we go lower. While it may be difficult to describe the way this *sounds* to us, this observation is the reason for the common way of voicing chords with large intervals near the bottom and smaller ones near the top.

Relation to Equal Temperament (et_beats.r)

A lucky accident is that when we divide the octave into 12 even pieces we get some very good approximations of perfect intervals.

Perfect Intervals			ET Intervals		
name	ratio	decimal	name	ratio	decimal
Perfect 5th	3/2	1.5	ET 5th	$2^{7/12}$	1.4983...
Perfect 4th	4/3	1.33...	ET 4th	$2^{5/12}$	1.3348...
"Perfect" Major 3rd	5/4	1.25	ET Major 3rd	$2^{4/12}$	1.2599...
"Perfect" Minor 3rd	6/5	1.2	ET Minor 3rd	$2^{3/12}$	1.1892...

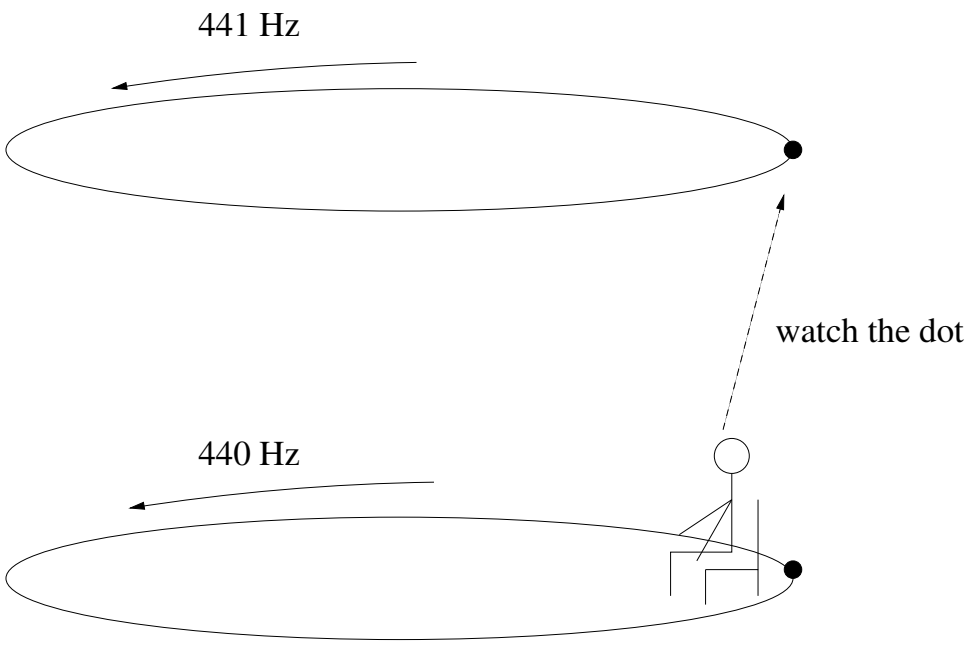
It is interesting to ponder how music would have developed if there were no number, such as 12, having this property. I am sure music would have managed, somehow, but the properties of twelve do seem fortunate for music's sake.

The program **et_beats.r** can be used to compare, side by size, perfect intervals with their equal tempered relatives. A careful look at this program will show, that the "waveform" is a little different from a sine wave. We will come back to this choice soon.

You will note the "beats" that occur with the equal tempered intervals.

The Phenomenon of Beats (beats.r)

We have observed "beats" when pitches are close to , but not exactly, simple integer ratios. Why is this? Consider the figure below:



Beats

Suppose we hear sines at 440Hz and 441Hz at the same time. Imagine we sit on a wheel going around at 440 Hz looking up at another wheel going around at 441 Hz. We sit at the location marked with a dot, and follow the motion of the corresponding dot on the other wheel. In this case we will see the dot going around at a rate of 1 Hz. Now think of the sine waves as the vertical distance of the dots from the origin.

- When dot is directly overhead the sine waves are equal and *reinforce* each other. Sound gets louder.
- When dot is opposite sine waves are opposite and *cancel* each other. Sound gets softer.

We will hear 1 Hz alternation between louder and softer = beats.

With sine waves will hear beats at the rate of the difference between frequencies (441-440 = 1 Hz), though the analysis is a bit more complicated for other periodic sounds.

Aliasing

Suppose we sample a sine wave at f Hz ($\sin(2\pi ft)$) at the sampling rate SR. The sample times are $\frac{1}{\text{SR}}, \frac{2}{\text{SR}}, \frac{3}{\text{SR}}, \ldots,$ so we get

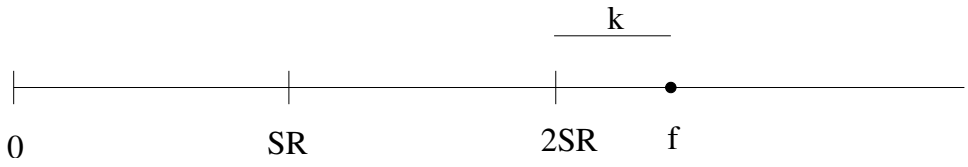
$$\sin(\frac{(2\pi f)1}{\text{SR}}), \sin(\frac{(2\pi f)2}{\text{SR}}), \sin(\frac{(2\pi f)3}{\text{SR}}), \ldots$$

True or False: When we play these at sampling rate SR we hear tone at f Hz? **Answer:** Sometimes.

1. Any frequency f can be written as

$$f = n\text{SR} + k$$

where $n \in \{\ldots, -2, -1, 0, 1, 2, \ldots, \}$ and $0 \leq k < \text{SR}$, as in the figure below



With f written this way the sampled values are

$$\sin\left(\frac{2\pi(n\mathrm{SR}+k)1}{\mathrm{SR}}\right), \sin\left(\frac{2\pi(n\mathrm{SR}+k)2}{\mathrm{SR}}\right), \sin\left(\frac{2\pi(n\mathrm{SR}+k)3}{\mathrm{SR}}\right), \ldots,$$

or, since the sine is 2π periodic,

$$\sin(\frac{2\pi k1}{\mathrm{SR}}), \sin(\frac{2\pi k2}{\mathrm{SR}}), \sin(\frac{2\pi k3}{\mathrm{SR}}), \ldots,$$

so $f = n\mathrm{SR} + k$ sounds the same as the frequency k:

$$n\mathrm{SR} + k \Longleftrightarrow k$$

2. For $f = SR - k$ get

$$\sin\left(\frac{2\pi(\mathrm{SR}-k)1}{\mathrm{SR}}\right), \sin\left(\frac{2\pi(\mathrm{SR}-k)2}{\mathrm{SR}}\right), \sin\left(\frac{2\pi(\mathrm{SR}-k)3}{\mathrm{SR}}\right), \ldots,$$

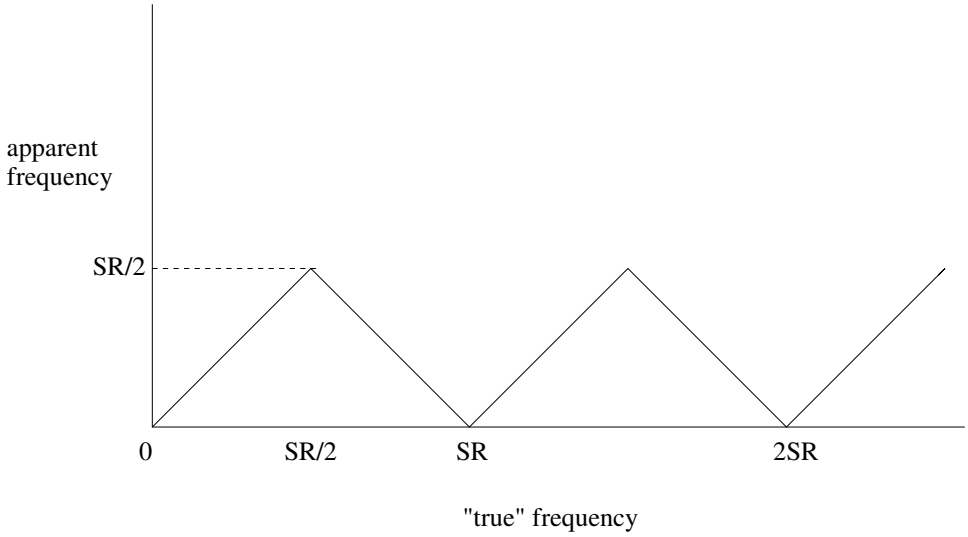
or

$$\sin(\frac{2\pi(-k)1}{\mathrm{SR}}), \sin(\frac{2\pi(-k)2}{\mathrm{SR}}), \sin(\frac{2\pi(-k)3}{\mathrm{SR}}), \ldots,$$

so

$$\mathrm{SR} - k \Longleftrightarrow -k \underbrace{\Longleftrightarrow}_{\sin(-x)=-\sin(x)} k$$

Thus the *apparent* frequency is given by



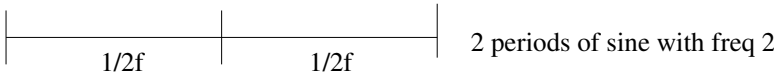
Consequence: Can't produce freq greater than SR/2 = "Nyquist" freq.

Since human hearing goes up to about 20 kHz a sampling of about 40 kHz (or 44.1 kHz) can represent the range of frequencies we can hear.

Sum of Sins and Periodicity (sum_of_sines.r, rand_timbre.r)

If we add 2 sines waves differing by octave we get

$$g(t) = a_1 \sin(2\pi f t + \phi_1) + a_2 \sin(2\pi 2 f t + \phi_2)$$



1. Both sines are $1/f$-periodic.
2. Sum of $1/f$-periodic functions is $1/f$-periodic.

so $g(t)$ is $1/f$-periodic. What do we hear?

1. Since the *period* of $g(t)$ is $1/f$ we hear freq f.
2. Since the *waveform* is changed, we hear different tone color or *timbre*.

This is demonstrated in **sum_of_sines.r**. This idea generalizes to

$$g(t) = a_1 \sin(2\pi f t + \phi_1) + a_2 \sin(2\pi 2 f t + \phi_2) + \ldots + a_n \sin(2\pi n f t + \phi_n)$$

Here, by similar reasoning, we still hear frequency f. This is demonstrated with the program **rand_timbre.r**. This program creates sums of sine waves whose frequencies are integer multiples of some "base" frequency f. The amplitudes of the sine waves are randomly chosen, as are the phases. You will hear that the sounds are all at the same pitch, while the timbre changes.

Returning to the **et_beats.r** program, recall that we observed beats for some "imperfect" intervals, such as the imperfect 5th associated with the ratio $2^{7/12}$. In this example we create our tones by summing together sine waves at integral multiples of the fundamental freqency, as described above. In the case of the imperfect 5th, since the ratio is *approximately* 3/2, the 2nd harmonic of the higher note is almost the same frequency as the 3rd harmonic of the lower note. Thus we will hear beating between these sine waves.

Glissando

The 1st homework asked you to create a "pitch vibrato," requiring a smooth transition over a range of frequencies. How do we do this?

We have observed that $f(t) = \sin(2\pi f t)$ is a constant frequency at f Hz.

Q: What is constant about $\sin(2\pi ft)$?

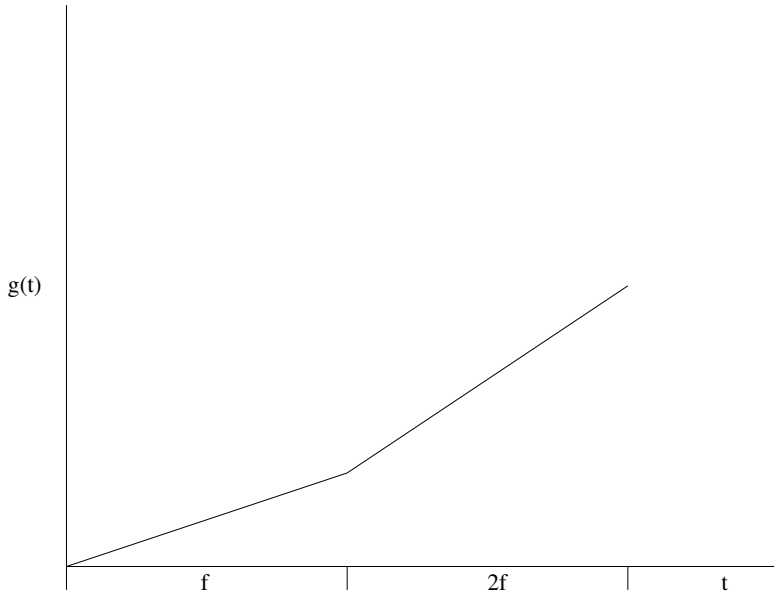
A: The "argument" to the sine, $2\pi ft$, has constant derivative:

$$\frac{d}{dt}2\pi ft = 2\pi f = \text{const.}$$

We see in the above case that for $\sin(g(t))$ the frequency at time t, $f(t)$, is given by

$$f(t) = \frac{\frac{d}{dt}g(t)}{2\pi}$$

as described in the following figure:



If, for $g(t)$ in this figure we take $\sin(g(t))$ the frequency will double (jump an octave) when the rate of change (slope) of $g(t)$ doubles.

The above equation holds more generally, however. For any function $g(t)$, the sound $\sin(g(t))$ will be a sine wave whose pitch at time t is $f(t)$ as above. Thus, if we want to create a function $\sin(g(t))$ that has a certain time-varying pitch, $f(t)$, we just need to "undo" this equation by integrating:

$$g(t) = 2\pi \int_0^t f(\tau)d\tau$$

Pitch Vibrato (pitch_vibrato.r)

Suppose we want our frequency function to be

$$f(t) = f_0 + \Delta \sin(2\pi vt)$$

as would be appropriate for pitch vibrato. Here f_0 is the base frequency we will move around, v is the rate of the vibrato in Hz., and Δ is the width of the vibrato — that is the pitch will change in the range $f_0 \pm \Delta$. Our sound

function will be $\sin(g(t))$ where we get $g(t)$ by integration as above:

$$g(t) = 2\pi(f_0 t - \frac{\Delta}{2\pi v}\cos(2\pi v t)) = 2\pi f_0 t - \frac{\Delta}{v}\cos(2\pi t)$$

Thus our pitch vibrato is

$$s(t) = \sin(2\pi f_0 t - \frac{\Delta}{v}\cos(2\pi v t))$$

We can try this out with the program **pitch_vibrato.r**.

Smooth Glissando (linear_freq.r)

Suppose we want a sine tone with a linear increase in frequency:

$$f(t) = rt$$

To create this we want $\sin(g(t))$ where we get $g(t)$ by integrating the frequency function times 2π. That is,

$$g(t) = 2\pi\frac{r}{2}t^2$$

so our sound function will be $s(t) = \sin(2\pi\frac{r}{2}t^2)$. We can hear this sound with the program **linear_freq.r**, though it would be a good idea to try to predict what we will here first. The actual sound departs from expectation in two ways:

1. While the frequency increases at a constant rate, r, the musical pitch does not increase at a constant rate. This is because our perception of pitch difference is "logarithmic" — that is, to have constant *pitch* change, the frequency much change by a constant *factor* for each unit of time.

2. Of course, the pitch will not continue to go up and up, due to the aliasing issue we have introduced. Rather we will observe the triangular up and down motion frequency as we say when we first introduced the idea of aliasing.

Uniform Glissando (uniform_gliss.r)

Suppose we want to create a glissando that increases at a uniform rate in musical terms. To be definite, say we begin at frequency f_0 at time $t = 0$ and rise one half step per second. In that case our frequency function, $f(t)$, will have

$$\begin{aligned} f(0) &= f_0 2^{0/12} \\ f(1) &= f_0 2^{1/12} \\ f(2) &= f_0 2^{2/12} \\ \vdots &\ \vdots\ \vdots \end{aligned}$$

To interpolate between these points in a smooth way we take

$$f(t) = f_0 2^{t/12}$$

which has *exponential* increase in frequency. More generally, say we wanted our frequency to traverse an octave every T seconds. The our frequency function would be

$$f(t) = f_0 2^{t/T}$$

Using the usual integration argument, the associated sine function would be

$$\sin(g(t)) = \sin(\frac{2\pi f_0 T 2^{t/T}}{\log 2})$$

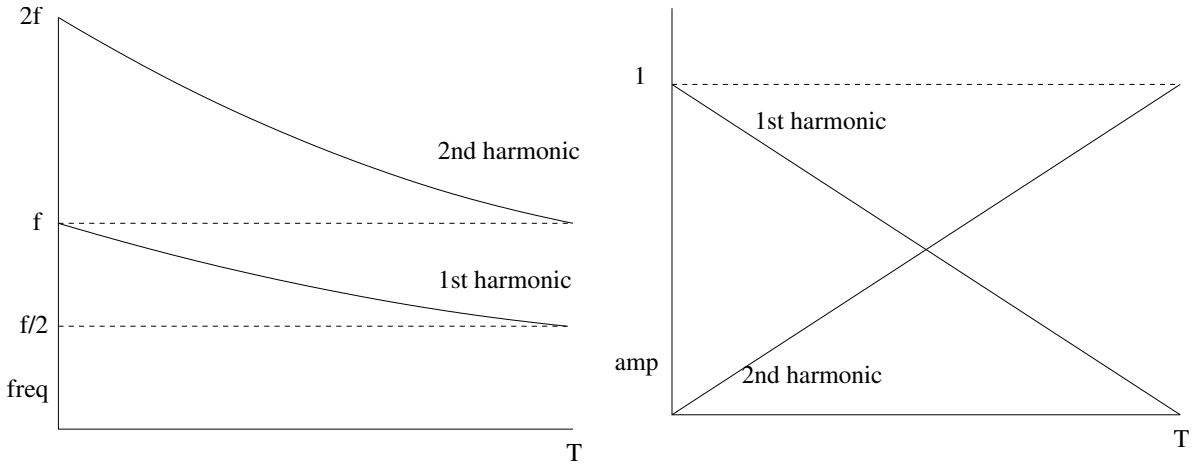


Figure 1.1: The evolution of frequencies (left) and amplitudes (right) of the two harmonics in the never-ending glissando.

Never-Ending Glissando (never_ending_gliss.r)

Suppose we construct a sound made up of two harmonics beginning at frequencies f_0 and $2f_0$ as in the figure. Over the course of T seconds both harmonics will drop one octave while retaining their precise 2:1 ratio. Thus, rather than appearing as two different sounds, we will hear a single sound whose timbre is richer than that of a sine wave. As this happens the amplitudes of these harmonics will also change, with the 1st harmonic decreasing linearly from 1 to 0 while the 2nd harmonic increase from 0 to 1. The resulting sound will be given by $h_1(t) + h_2(t)$ where

$$\begin{aligned} h_1(t) &= (1 - t/T)\sin(\frac{-2\pi T f_0 2^{-t/T}}{\log 2}) \\ h_2(t) &= (t/T)\sin(\frac{-2\pi T 2 f_0 2^{-t/T}}{\log 2}) \end{aligned}$$

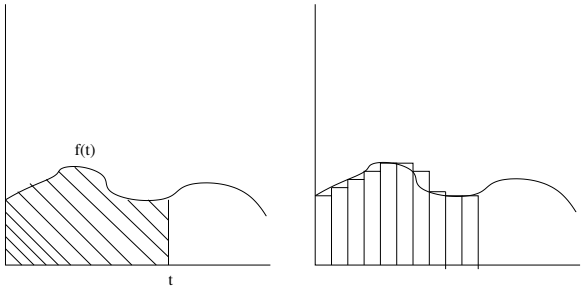
This situation is illustrated in Figure 1.1. Observe the following

1. At $t = 0$ we have a single sine wave at frequency f_0 with amplitude 1.
2. At $t = T$ we have exactly the same situation.

Thus we could concatenate any number of copies of $h_1(t) + h_2(t)$ to get a sound that appears to continually decrease without ever becoming low.

Numerical Integration (random_pitch_walk.r)

All of this integration can be problematic since most functions cannot be integrated in closed form. However, it is easy to perform the discrete analog of integration for any sampled function (i.e. everything we treat in this class).



Consider the left panel of the figure above. Integration of $f(t)$ requires us to compute the area under the the curve of $f(t)$, up to t, as a function of t:

$$F(t) = \int_0^t f(\tau)d\tau$$

In our case, we only have a *sampled* version of our function:

$$f(0\Delta), f(1\Delta), f(2\Delta), \ldots$$

where $\Delta = 1/SR$. If we knew nothing about the values between our sample points, a simple approximation would assume that f is constant between the samples, as shown in the right panel of the figure. Such a function is easy to integrate, simply by adding the areas of rectangles. We get

$$\begin{aligned} F(1\Delta) &= \Delta f(0\Delta) \\ F(2\Delta) &= \Delta f(0\Delta) + \Delta f(1\Delta) \\ F(3\Delta) &= \Delta f(0\Delta) + \Delta f(1\Delta) + \Delta f(2\Delta) \\ &\vdots \quad \vdots \quad \vdots \end{aligned}$$

or more generally

$$F(n\Delta) = \sum_{m=0}^{n-1} f(m)\Delta = \Delta \sum_{m=0}^{n-1} f(m)$$

R has a function that performs exactly this operation known as **cumsum** (for cumulative sum). For a vector f, cumsum(f) is the vector whose nth component is the sum of the first n elements. Thus Δcumsum(f) will function as our integral for f. Now, for any sampled pitch function, $f(t)$, we can produce a sine wave with the pitch $f(t)$ by

$$y = \sin(2\pi\Delta\text{cumsum}(f))$$

The program **rand_pitch_walk.r** shows an example of this technique for a randomly created pitch function.

Chapter 2

Fourier Analysis

We have seen that

$$g(t) = a_1 \sin(2\pi f t + \phi_1) + a_2 \sin(2\pi 2 f t + \phi_2) + \ldots + a_n \sin(2\pi n f t + \phi_n)$$

is periodic with period $1/f$ so $g(t)$ sounds at f Hz. As we varied the amplitudes $a_1, \ldots, a_n$ we saw that the timbre or color of the sound changed.

Amazing Fact #2

Any $\frac{1}{f}$-periodic function, $g(t)$, can be approximated as well as we like by

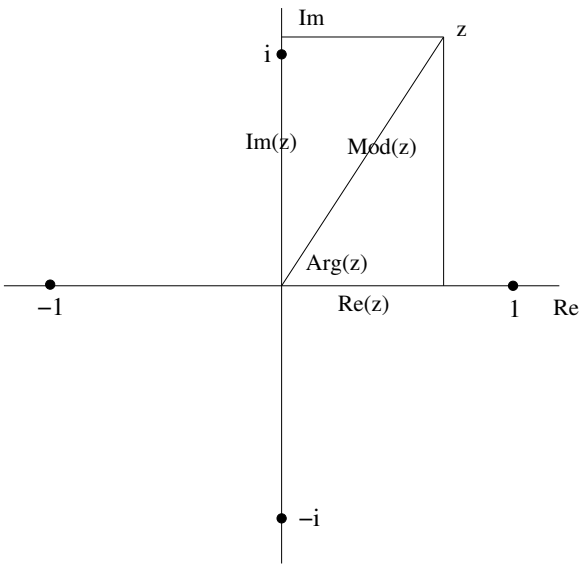
$$g(t) \approx a_1 \sin(2\pi f t + \phi_1) + a_2 \sin(2\pi 2 f t + \phi_2) + \ldots + a_n \sin(2\pi n f t + \phi_n)$$

by choosing the number of terms, n, and the coefficients, $a_1, \ldots, a_n$ and the phases $\phi_1, \ldots, \phi_n$ appropriately. In fact, if t takes only a finite number of values, $\frac{0}{SR}, \frac{1}{SR}, \ldots, \frac{N-1}{SR}$ then the approximating in the equation becomes an equality.

Q: How do we get the a's and the ϕ's?

A: Fourier Analysis

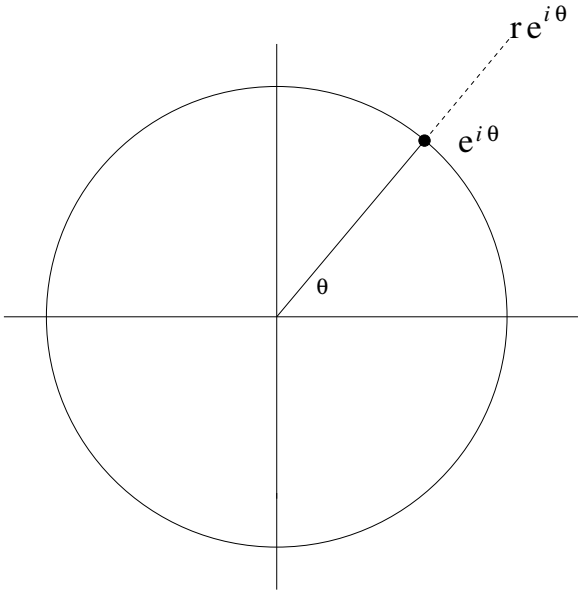
Complex Numbers



Complex numbers are numbers $z = a + bi$ where $i = \sqrt{-1}$. Students first encountering complex numbers often question the use of such numbers which don't seem to represent anything familiar in the world we know. Just as negative numbers have natural interpretations in terms of debt (having a negative amount of money) or position (+ in one direction and - in the opposite), complex numbers have concrete interpretations in some cases. One of the most compelling of these is the subject we are about to study.

Complex numbers can be visualized as points in the complex plane where the horizontal axis denotes the real part of the number and the vertical axis denotes the imaginary part. That is, if $z = a + bi$ is a complex number then the point with coordinates (a, b) is the graphical representation of z. A complex number has 4 attributes that will be important to us. The "real part" of z is denoted by $\mathrm{Re}(z)$. With z as above $\mathrm{Re}(z) = a$. Similarly the "imaginary part" of z, $\mathrm{Im}(z) = b$ (the imaginary part is a real number). A complex number, z makes an angle with the real axis which we denote as $\mathrm{Arg}(z)$ and has a length $\mathrm{Mod}(z) = \sqrt{a^2 + b^2}$. These quantities are illustrated in the figure above. The R program handles complex numbers and uses the same notation of Re,Im,Mod,Arg.

We will occasionally refer to the polar representation of a complex number. Suppose we define $e^{i\theta}$ as the number on the unit circle of the complex plane making angle θ with the real axis, as in the figure below. This may not seem like it has anything to do with the familiar view of the exponential function, so just take it as a definition for now. If we multiply this number by a real constant r, then $re^{i\theta}$ "points" in the same direction as $e^{i\theta}$ but has length r. In other words, if z is a complex number with $\mathrm{Mod}(z) = r$ and $\mathrm{Arg}(z) = \theta$, then $z = re^{i\theta}$. Sometimes this is called the "polar" representation of z, since it is essentially polar coordinates.



One can do many things with complex numbers that we are used to doing with real numbers. For instance, complex numbers can be added, just like two dimensional vectors. That is, if $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$ then $z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$. More unusual is multiplication for complex numbers, which is most intuitive in polar form. If $z_1 = r_1 e^{i\theta_1}$, and $z_2 = r_2 e^{i\theta_2}$ then $z_1 z_2 = r_1 r_2 e^{i(\theta_1 + \theta_2)}$. Note that the moduli are multiplied while the angles are added. Thus the familiar rule of $e^x e^y = e^{x+y}$ holds for complex numbers too.

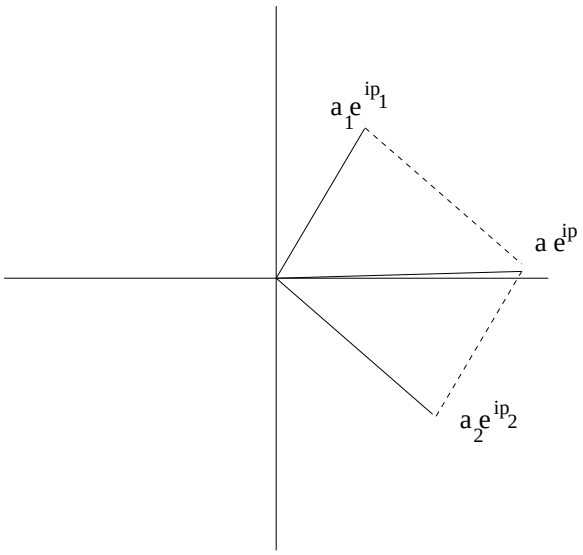
Adding Sines of Identical Frequency

Some kinds of calculations are easier with complex numbers. Here is an example. Suppose you have two cosines at the same frequency, f, but with different amplitudes and phases: $a_1(t) = a_1 \cos(2\pi f t + \phi_1)$ and $a_2 \cos(2\pi f t + \phi_2)$.

What is the result when we add these together. In polar complex notation we have

$$\begin{aligned} a_1\cos(2\pi ft+\phi_1)+a_2\cos(2\pi ft+\phi_2) &= \mathrm{Re}(a_1e^{2\pi ift+i\phi_1})+\mathrm{Re}(a_2e^{2\pi ift+i\phi_2}) \\ &= \mathrm{Re}(a_1e^{2\pi ift+i\phi_1}+a_2e^{2\pi ift+i\phi_2}) \\ &= \mathrm{Re}((a_1e^{i\phi_1}+a_2e^{i\phi_2})e^{2\pi ift}) \\ &= \mathrm{Re}(ae^{i\phi}e^{2\pi ift}) \\ &= a\cos(2\pi ft+\phi) \end{aligned}$$

In words, we add the complex numbers $a_1e^{i\phi_1}$ and $a_2e^{i\phi_2}$ associated with the amplitudes and phases of the original cosines. The result, expressed in polar form, gives the amplitude and phase of the resulting cosine. Note that the result has the same frequency f. See the figure below:



Finite Fourier Transform

Suppose we have a sampled function (a waveform) consisting of N points, $f(0), f(1), \ldots, f(N-1)$ where N is even. We will represent f as a sum of cosine waves that oscillate $0, 1, \ldots, N/2$ times over the N points. Examples of such cosine function are given in Figure 2.1. Note that the cosine that oscillates 0 times is just function that is identically equal to 1, while the cosine that oscillates $N/2$ times alternates back and forth between 1 and -1.

To be precise, we will use the functions $c_n(j)$ for $n = 0, 1, \ldots, N/2$ defined by

$$c_n(j) = \cos(\frac{2\pi nj}{N})$$

for $j = 0, 1, \ldots, N-1$. In representing f we will allow the shifting and scaling of these functions. That is, we let

$$c_{n,a_n,\phi_n}(j) = a_n\cos(\frac{2\pi nj}{N}+\phi_n)$$

for $j = 0, 1, \ldots, N-1$. Then we represent f as

$$\begin{aligned} f &= c_{0,a_0,\phi_0}+c_{1,a_1,\phi_1}+\ldots+c_{N/2,a_{N/2},\phi_{N/2}} \\ &= \sum_{n=0}^{N/2} c_{n,a_n,\phi_n} \end{aligned}$$

for some collection of amplitudes $\{a_n\}$ and phase shifts $\{\phi_n\}$. Pictorially, this is represented as follows

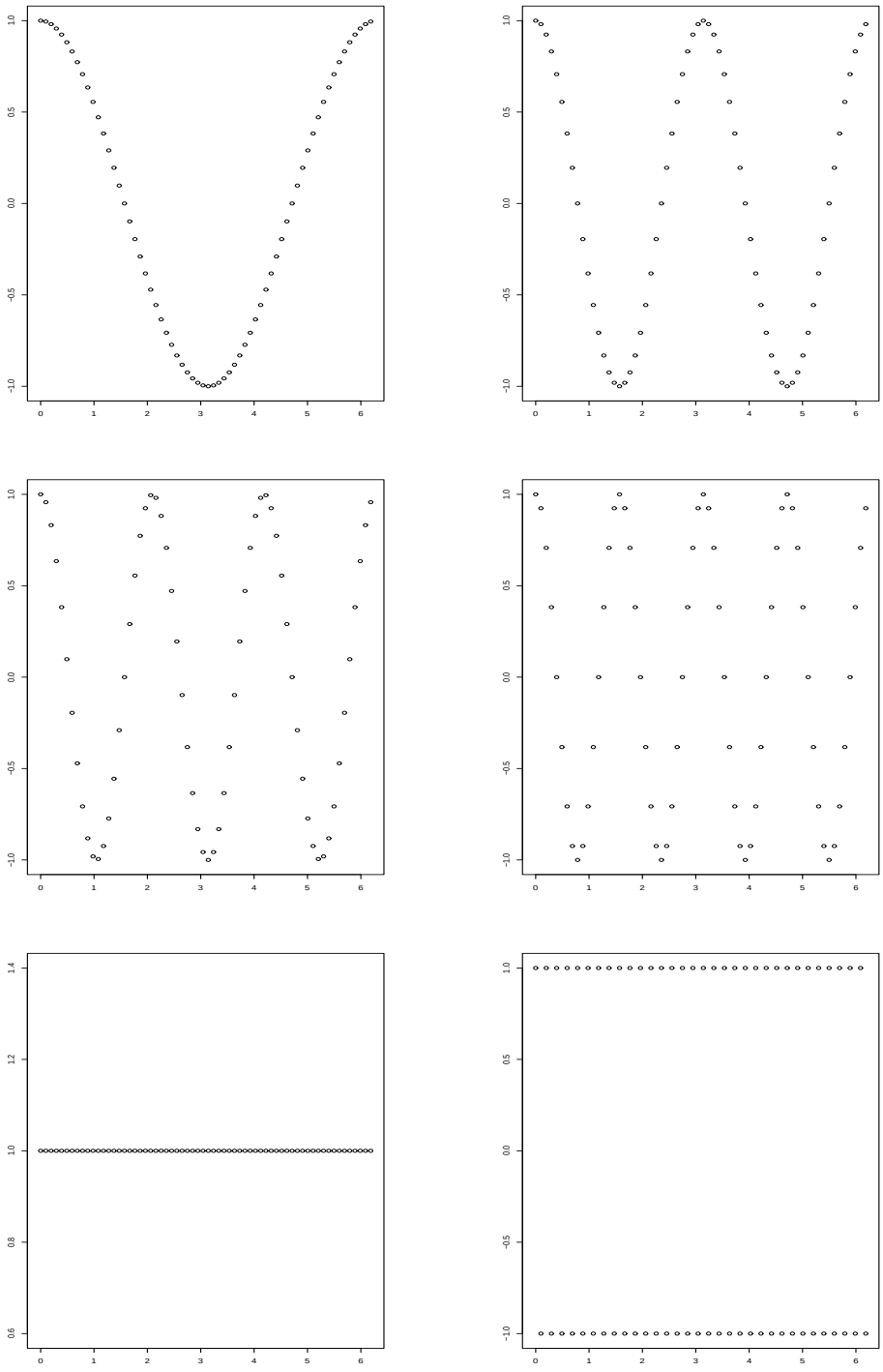
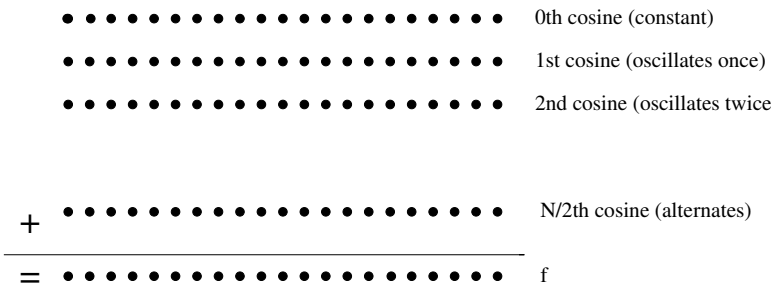


Figure 2.1: The discrete cosine vectors used in representing an N-point function (waveform). Pictures in "reading" order are the cosines that oscillate 1,2,3,4,0, and N/2 times. Note that the cosine that oscillates 0 times is just constant at 1, while the one the oscillates $N/2$ times oscillates back and forth between 1 and -1.



The key to getting this representation is the Finite Fourier Transform (FFT).

Definition of FFT

In this document we abbreviate the Finite Fourier Transform as FFT. "FFT" is also commonly used as an abbreviation for "Fast Fourier Transform" which is an *algorithm* that is used to implement the Finite Fourier Transform. Since both refer to the same mathematical construction, if there is any confusion it is usually harmless.

The FFT of $f = (f(0), \ldots, f(N-1))$, is a complex-valued N-vector $F = (F(0), F(1), \ldots, F(N-1))$ given by

$$F(n) = \sum_{j=0}^{N-1} f(j) e^{-\frac{2\pi i n j}{N}}$$

for $n = 0, 1, \ldots, N-1$. We will denote the relationship between a f and its finite Fourier transform, F, as

$$f \overset{\text{FFT}}{\longleftrightarrow} F$$

We will come back to this definition later, but first let's discuss its consequences.

Interpretation of FFT

If $f = (f(0), f(1), \ldots, f(N-1))$ and F is the FFT of f:

$$f \overset{\text{FFT}}{\longleftrightarrow} F$$

then we can recover the amplitudes and phases of the cosines used in our reconstruction of f. That is

$$\begin{aligned} \phi_n &= \text{Arg}(F(n)) \\ a_n &= \begin{cases} \frac{\text{Mod}(F(n))}{N/2} & \text{for } n = 1, \ldots, N/2 - 1 \\ \frac{\text{Mod}(F(n))}{N} & \text{for } n = 0, N/2 \end{cases} \end{aligned}$$

FFT in R

R computes the FFT of an N-vector, f, as

```
> F = fft(f);
```

Each of the N components of F is a *complex* number. R handles complex numbers in a wide variety of calculations, and nearly any calculation where complex numbers make sense such as addition, multiplication, etc. Usually, if one is going to plot the complex numbers corresponding to the FFT it is better to plot some real-valued attribute such as their amplitude, (**Mod(F)**), or phases, (**Arg(F)**). While R can compute an FFT for any length vector, the FFT algorithm works most efficiently when the length is a power of 2. We will always choose this length to be a power of 2, since it costs us nothing.

Due to the 1-based indexing of the R program, $F(n)$ for $n = 1, \ldots, N$ gives the phase and amplitude for the cosine that oscillates $n-1$ times in N points. This is yet another argument for the inferiority of 1-based indexing, but the issue is easy enough to deal with.

Understanding the Meaning of the FFT (sine_id_1.r,sine_id_2.r, sawtooth.r)

As a first example we will construct a function which we *know* consists of a single cosine using the R program **sine_id_1.r**. This program allows one to vary the frequency and phase of the cosine, though the frequency (in oscillations per N points) must be integral. When we compute the FFT of this one-cosine function, all values of the FFT are $0 = 0 + 0i$ except for the component corresponding to our cosine. That is, if our cosine is at frequency n, then only $F(n+1)$ will be nonzero. We then can recover the amplitude and phase as

```
> amp = Mod(F(n+1))/(N/2)
> ph = Arg(F(n+1))
```

The program **sine_id_2.r** does the same thing, but now we take a function formed as the sum of 3 such cosines, again with integral frequency, having different amplitude and phase. Looking at a plot of this function, as the program does, it is hard for us to imagine how it was constructed, but the FFT has no difficulty in "resolving" the function into its constituent cosines. Again the three FFT components "light up" and from these we can recover the amplitudes and phases as before.

The R program **sawtooth.r** shows a more interesting example. In this case we can take f to be any N-point function we want, such as a sawtooth waveform, or even something generated in a random fashion. The program loops through all possible frequencies, from 0, to $N/2$, and adds in the appropriate cosine wave for the current frequency, scaled and shifted as prescribed by the appropriate FFT component. Reading directly from the code, this is accomplished by

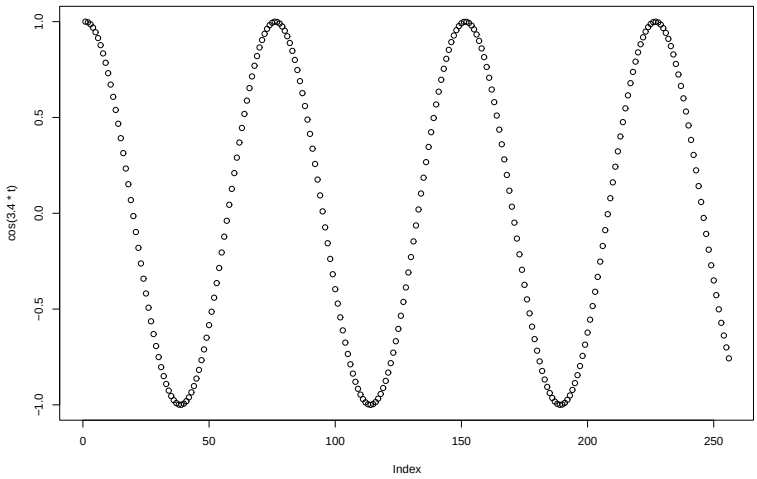
```
> yhat = yhat + Mod(Y[i])*cos((i-1)*t + Arg(Y[i]))  # add in new cosine
```

In this example **yhat** is the "accumulator" that holds the sum from the "currently visited" frequencies. In this line of code we add to the accumulator the scaled and shifted cosine which oscillates $i - 1$ times (remember the 1-based indexing of R). Each time through the loop we display the current approximation to our target function. As we move through the iterations we can see the approximation getting better and better until it is indistinguishable from the target.

The two examples in the program — the "sawtooth" function and the randomly generated function show two different views of the FFT. In the first example we take a periodic function which is a "sawtooth" or up and down triangle shape. We use the FFT to represent a single period of this function in terms of sine waves that oscillate integral numbers of times over the period. For those familiar with the idea of the "Fourier Series," this is a discrete approximation of the series. The next example takes a randomly generated function. It seems rather odd to think of this as a period of pitched sound, though we could view it that way. If we regard it as a generic wave form (not necessarily periodic), we see that we can still approximate it in terms of cosines. Thus, the FFT works as a decomposition for any discrete function, such as a sound file.

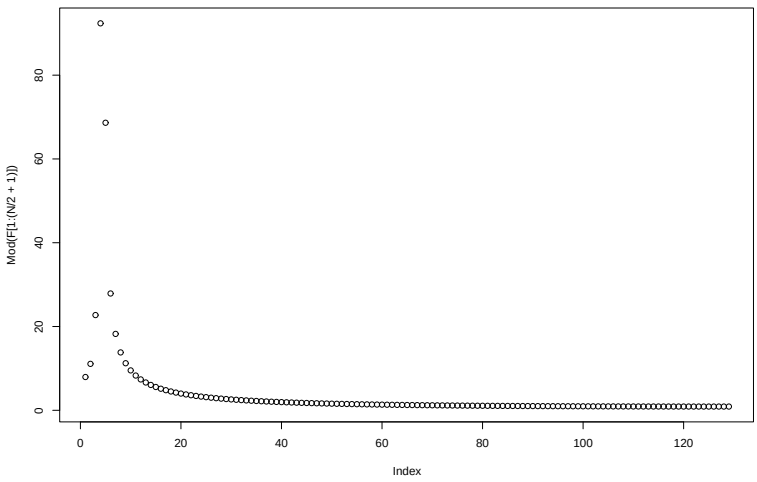
Arbitrary Sine Waves (arbitrary_sine.r)

The FFT represents a waveform in terms of cosines that oscillate an *integral* number of times per FFT-length. What if we analyze a sine or cosine that doesn't "match" any of our FFT frequencies, as in the following figure? Note that, over the 256 points we will examine (our FFT length) the sine wave ends up with a different phase than it starts with.



In this case it is hard for the FFT to approximate the function well since none of the FFT "building blocks" look like

the function we approximate. Of course, since the FFT can represent anything, the result is that more cosine waves are needed to accurately represent this function. The following figure shows the resulting modulus of the FFT for this sine wave. Note that we need contributions from many sine waves whose frequencies are in the neighborhood of the one we wish to capture.

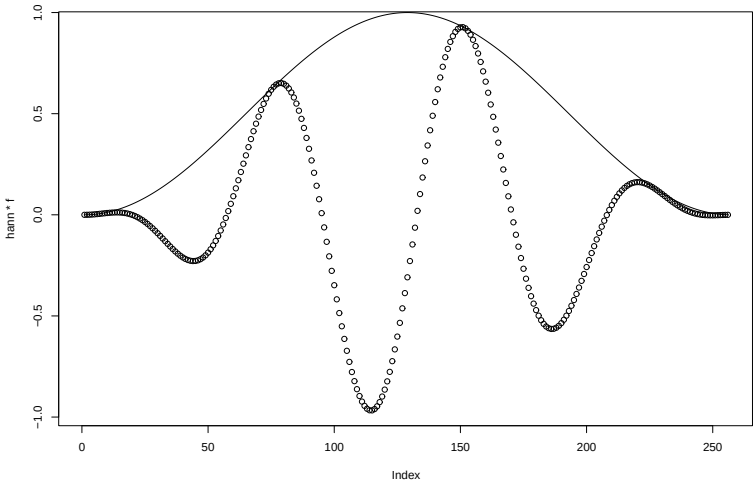


Whether or not this phenomenon is a *problem* depends on what one is trying to do. At present, we look to the FFT to provide the frequency content of the input signal. Thus, if the signal is a single sine or cosine, we would

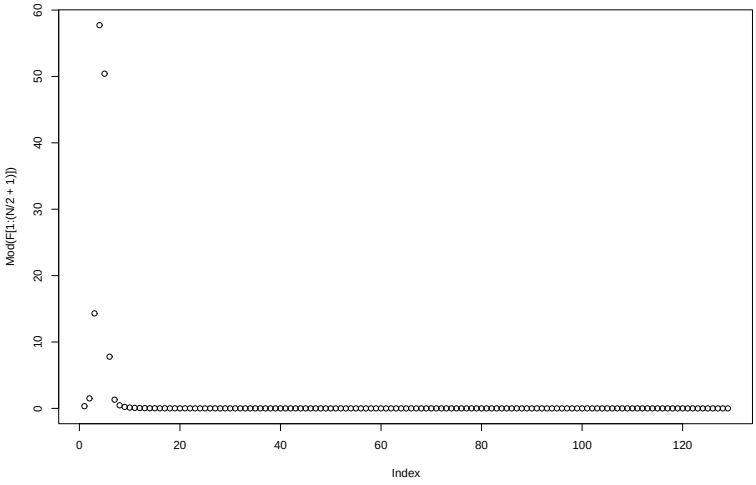
like to see this more accurately reflected in the FFT. We can accomplish this by *windowing* the data — multiplying our input by a function that trails off to zero at both endpoints. There are many possible choices of windows and a huge amount of literature on the effects of this choice. This topic will not concern us in this class, so we will use the common "Hann," or "raised cosine" window for all of our experiments. This window is given by

$$h(j) = \frac{1 + \cos(\frac{2\pi j}{N} - \pi)}{2}$$

$j = 0, \ldots, N-1$. and is depicted below. Superimposed on the figure is the effect of multiplying the signal considered above with the Hann window.



Finally, we show below the effect on the FFT. Note that the frequency content is less spread out, which is more consistent with our perception of this signal.



Usually real-audio samples are windowed before Fourier transforms are taken.

FFT of Pitched Musical Sound (spectrum_movie.r)

Suppose we have a musical instrument note at f Hz., sampled at SR samples per second. We take N samples from this sound and compute the FFT,

$$y \overset{\text{FFT}}{\longleftrightarrow} Y$$

What "bins" of the FFT will have energy — that is what values of n will have $|Y(n)|$ significantly greater than 0?

Remember that the FFT bins, indexed by $0, 1, \ldots, N/2$, correspond to the number of oscillations per FFT-length — that is, the number of oscillations per N points. Thus

$$\text{bin } 1 \Longleftrightarrow 1 \text{ osc. per } N \text{ points} \Longleftrightarrow 1 \text{ osc. per } \tfrac{N}{\text{SR}} \text{ secs.} \Longleftrightarrow \tfrac{\text{SR}}{N} \text{ Hz.}$$

and more generally

$$\text{bin n} \Longleftrightarrow \text{n osc. per } N \text{ points} \Longleftrightarrow \text{n osc. per } \tfrac{N}{\text{SR}} \text{ secs.} \Longleftrightarrow n\tfrac{\text{SR}}{N} \text{ Hz.}$$

So the Hz, f, for bin n is

$$f = n\frac{\text{SR}}{N}$$

and the bin, n, for freq f Hz. is

$$n = f\frac{N}{\text{SR}}$$

Note that $f\frac{N}{\text{SR}}$ is not necessarily an integer, so the frequency energy will cluster *around* $f\frac{N}{\text{SR}}$.

Recall that periodic (pitched) signals can be represented as sums of sines or cosines, at integer multiples of the fundamental frequency. Thus we expect to see energy at or around bins

$$f\frac{N}{\text{SR}}, 2f\frac{N}{\text{SR}}, 3f\frac{N}{\text{SR}}, \ldots$$

In the program **spectrum_movie.r** we examine the frequency spectra of a collection of audio "frames." In using the term *frames* we make an analogy with a movie in which a frame captures a snapshot of a visual scene. Similarly, and audio frame can be viewed as a snapshot of an aural scene. Thus, we compute a sequence of FFTs corresponding to neighboring "blocks" of audio data. The audio data for this example are a sequence of octaves played on the oboe starting from "low B♭" right below middle C. Since this note is one half step above the octave below A = 440 Hz., the frequency must by $f = 220 \times 2^{1/12}$. Thus the frequency bin where this note will appear is

$$n = f\frac{N}{\text{SR}} = \frac{220 \times 2^{1/12} \times 512}{8000} \approx 15$$

The **spectrum_movie.r** program will verify that this is true. This is the first time we have made any use of actual recordings in our class, though we will do this quite a bit in the weeks to come.

Modeling Pitched Sound (timbre_copy.r)

Since pitched sounds are nearly periodic, they can be modeled as a sum of sine waves with frequencies at integral multiples of the "fundamental" (the frequency we hear):

$$g(t) = a_1 \sin(2\pi f t) + a_2 \sin(2\pi 2 f t) + \ldots + a_n \sin(2\pi n f t)$$

where the amplitudes are estimated from real data. You will notice that we have left the phase shifts out of the above representation. This is because they are not perceptually relevant for a pitched sound — in the situation of the above equation, you will hear the same thing for any choice of phases. To estimate the amplitudes $a_1, a_2, \ldots, a_n$, we will just look at the actual modulus of the FFT of an actual recorded sample at the frequencies $f, 2f, \ldots, nf$.

The estimation of the amplitudes is done in the program **timbre_copy.r**. In this program, we read in the oboe data, given on the web site, and produce a replica of the oboe timbre by using harmonic amplitudes estimated from these data. When we deal with real sound, we may know the name of the note, (A,B♭,C♯, ...), though the frequency

is only known approximately. Thus, the program looks at a range of FFT "bins" in the neighborhood of the position where each harmonic (multiple of the fundamental f) should lie. For the kth harmonic, we estimate a_k to be

$$\hat{a}_k = \max_{n \in N(kf)} |G(n)|$$

where $N(kf)$ is a neighborhood of the kth harmonic ($\pm$ 3 bins in the program) and G is the FFT of our sound g. We would get similar sounding results if we looked at the sum of the $|G(n)|$ over each neighborhood, or, perhaps, $\sqrt{\sum |G(n)|^2}$

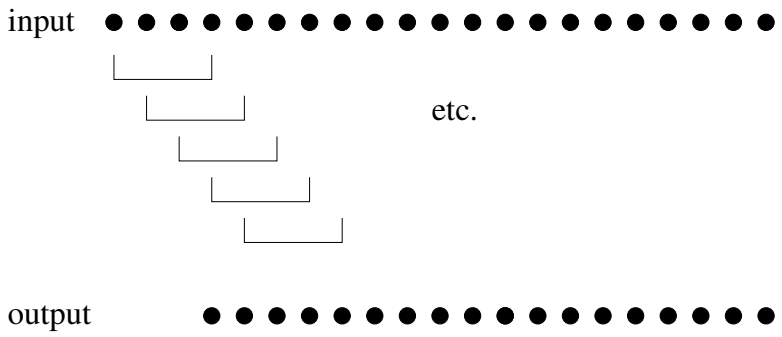
In performing this estimation we have created a simple *model* for the data. Model-based approaches are flexible: we can apply this model to other notes, produce notes of any length, add vibrato and glissando, etc. Consider this approach as compared to one that works directly with the sampled sound. A purely sample-based approach is much more limited, since it can only play the exact samples back (with minor modifications) and can thus create only a small range of variation.

Noise (white_noise.r, colored_noise.r)

We can think of *noise* as sound that is not pitched, and thus has no "singable" tone — more formal definitions are possible, of course, nearly all involving some notion of randomness in the data. The Fourier transform can also be used to capture the perceptual characteristics of noisy sounds, as well as pitched sounds.

When samples are chosen randomly and *independently* the result is often deemed "white" noise. While we will not define "independence" in a formal sense here, we mean that each sample is chosen without regard for the other samples, as would be the case with a random number generator. The term "white" is used in analogy with light. When all frequencies of light are present, as in light that comes from the sun, the light looks white in color. We can see from the program **white_noise.r** that independent random numbers produce energy at at all frequencies, though the contribution at each frequency is random. The program also plays the associated sound, which sounds something like the ocean, or a huge volume of rushing water.

Anything we do to *correlate* the samples (introduce ways in which the values of some samples influence the values of other samples), results in what is called "pink" noise, or correlated noise. The simplest way we could achieve this would be to *average* adjacent samples of white noise as in the following figure:



Such a scheme is called a *moving average*. The term moving average is used even if we do not *average* a collection of samples, but rather, more generally, if we take some linear combination of adjacent samples. That is, if $x_1, x_2, \ldots$ form our initial sequence, the moving average is given by

$$y_j = \sum_{k=-K}^{+K} \alpha_k x_{j-k}$$

For instance, in the case of the example shown in the figure we have

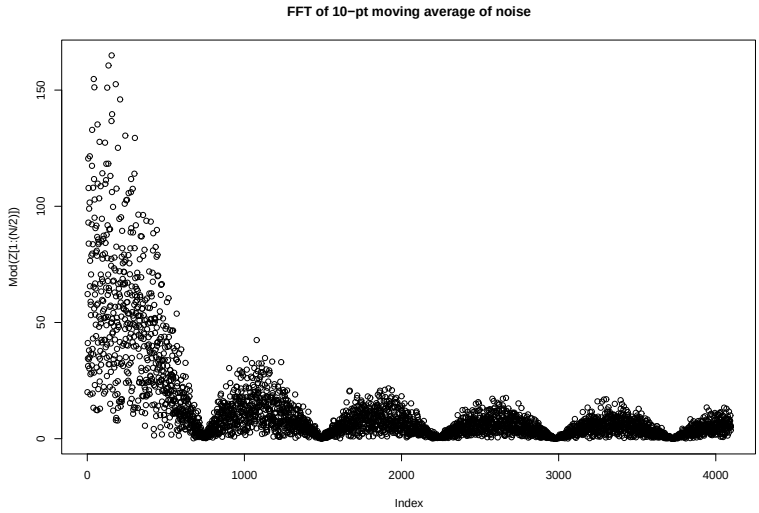
$$y_j = \sum_{k=0}^{3} \frac{1}{4} x_{j-k}$$

We could compute this moving average in R by setting

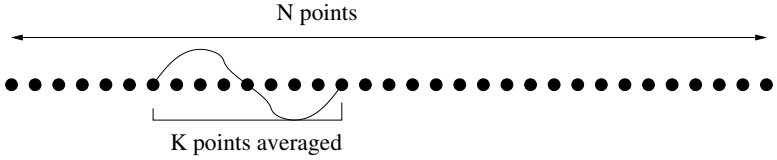
```
> N = 8192
> x = rnorm(N)
> y = rep(0,N)
> for (j in 1:N-4)  y[j] = mean(x[j:(j+4)]);
```

Note the difference both in sound and in the appearance of the FFT in the data produced by the **colored_noise.r** program. Observe that the FFT still captures the perceptual characteristics of the sound, such as "diminished high frequencies."

The **colored_noise.r** program creates sound by averaging together any number of adjacent samples. As we experiment with this program, the first thing we notice is that the quality of the sound changes as we average in longer windows of samples. This perceptual change is indicated in the FFT, where we see that the some of the higher frequencies are diminished, though the shape of the spectrum depends on the length of the averaging window.



We also see an unusual pattern emerging with the FFT as depicted above: as we average together K adjacent samples, certain frequencies completely disappear from the resulting FFT spectrum. What is going on here? If we consider the figure below, we see that the moving average with K points will *annihilate* any sine wave with a period of K points. This is because we get 0 if we average up an entire period of the sine. This same argument holds for sine waves having periods of length $K, K/2, K/3, \ldots$. These periods correspond to $N/K, 2N/K, 3N/K, \ldots$ oscillations per N points — the frequency unit of the FFT. These are the places where we see the "valleys" in the FFT. For instance, in the figure above we used an 8192-pt FFT with an average of 10 consecutive points, giving valleys at multiples of about 819.



Filtering

Suppose we have two vectors, a "long" one, $x = (x_0, x_1, \ldots, x_{N-1})$ and a "short" one $a = (a_0, \ldots, a_L)$. We can use the vector a to generate a new sequence from x called y, by letting

$$y_j = a_0 x_j + a_1 x_{j-1} + a_2 x_{j-2} + \ldots, a_L x_{j-L}$$

A simple example of this is the "averaging" operation we have already seen. In this case $a = (\frac{1}{L+1}, \frac{1}{L+1}, \ldots, \frac{1}{L+1})$. This operation is known as filtering, convolution, or moving average. As notation for this operation we will write

$$y = a * x$$

where

$$\begin{aligned} (a * x)(j) &= \sum_{l=0}^{L} a_l x_{j-l} && (2.1) \\ &= \sum_{(l,k):l+k=j} a_l x_k && (2.2) \end{aligned}$$

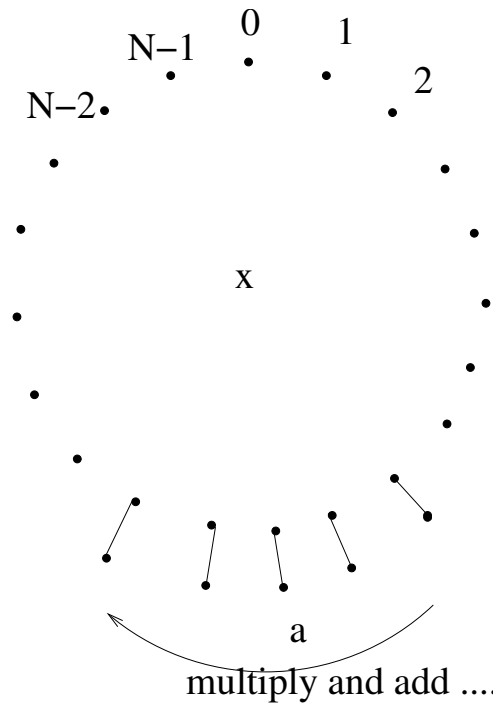
The filtering operation has an interesting relation to the FFT, as follows:

Amazing Fact # 3

If we are careful in the definition of the filtering operation, the effect of *filtering* in the "time domain" by a is that of *multiplying* by the FFT of a in the "frequency domain." That is, if $x \overset{\text{FFT}}{\longleftrightarrow} X$ and $a \overset{\text{FFT}}{\longleftrightarrow} A$ then

$$a * x \overset{\text{FFT}}{\longleftrightarrow} A \cdot X \tag{2.3}$$

A little care is needed in interpreting this equation. Since we are dealing with the *finite* Fourier transform here, we must be more precise about the meaning of the convolution. In particular, we need a "circular" convolution. The definition of the circular convolution is almost identical to the one already presented. The difference is that we interpret the addition and subtraction in 2.1 and 2.2 as being *modulo* N (remainder when divided by N), where N is the length of the sequences. In these equations we treat a also as an N-length sequence where most of the components of a may be 0. While it is circular convolution that transforms exactly into the product of the Fourier transforms, this is approximately true if we don't bother with the circular aspect of the convolution.



While we don't emphasize the mathematical aspects of this course, the convolution relation of Eqn. 2.3 is easy to verify as follows:

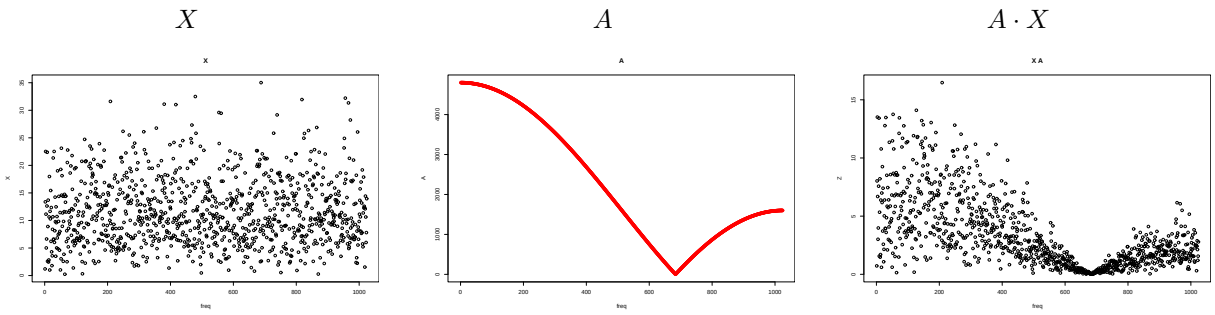
$$\begin{aligned}
\mathrm{FFT}(a*x)(n) &= \sum_{j=0}^{N-1}(a*x)_j e^{\frac{-2\pi ijn}{N}} \\
&= \sum_{j=0}^{N-1}\sum_{l+k=j} a_l x_k e^{\frac{-2\pi ijn}{N}} \\
&= \sum_{l=0}^{N-1}\sum_{k=0}^{N-1} a_l x_k e^{\frac{-2\pi iln}{N}} e^{\frac{-2\pi ikn}{N}} \\
&= \sum_{l=0}^{N-1} a_l e^{\frac{-2\pi iln}{N}} \sum_{k=0}^{N-1} x_k e^{\frac{-2\pi ikn}{N}} \\
&= A(n)X(n)
\end{aligned}$$

Understanding Filtering (noise_filter.r)

Let's consider applying the averaging filter

$$a = \underbrace{(\frac{1}{L}, \frac{1}{L}, \ldots, \frac{1}{L})}_{L \text{ components}}$$

to some white noise, x, as before, with $x \overset{\mathrm{FFT}}{\longleftrightarrow} X$. We have already seen that white noise has a flat (and random) spectrum, as in the left panel of the figure below. The middle panel of the figure shows the modulus of A — the transform of our filter, a. In constructing A we simply "pad" a with zeros to make it the same length as x before transforming — this corresponds to the identical filter operation, but leaves A comparable with X. We have learned the effect of filtering (convolving) the noise, x, with our averaging filter, a, *in frequency domain* is the *product* $A \cdot X$, as given in the right panel of the figure.



The Fourier domain gives an intuitive understanding of the effects of our averaging filter, as follows. The filter attenuates or enhances frequencies according to the value of the modulus of A at the frequency. Thus, one particular frequency is completely annihilated by filtering with a, while frequencies in this neighborhood are considerably diminished. On the other hand, the filter has negligible effect on the low frequency components of our signal. The program **noise_filter.r** performs the calculations described above, as well as playing audio both before and after applying the averaging filter.

Creating a Filter (lowpass_filter.r)

The previous example leads to an interesting idea. Suppose we begin with a precise description of what we want our filter to do, in terms of its "frequency response" — the way it attenuates or enhances the various frequencies. Such a description is given by the middle panel of the figure above. For example we may want:

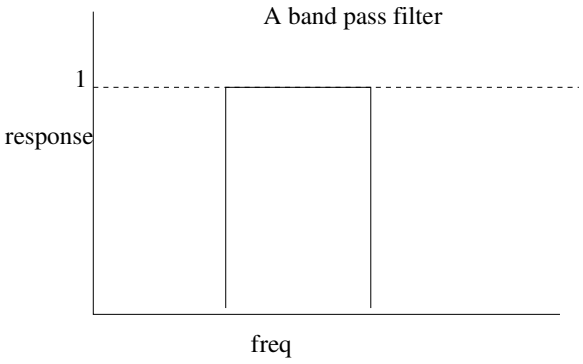
A low pass filter: let only frequencies through that are less than some specified cutoff.

A high pass filter: let only frequencies through that are above than some specified cutoff.

A band pass filter: let only frequencies through that are in some specified range.

We can create such a filter using the following procedure.

1. Create the desired frequency response, such as the band pass filter below.



2. Suppose we can find the filter coefficients that transform to this response (more coming on this). If this filter is given by a, we have seen that convolution with a is multiplication by the desired response. Thus a will be the desired filter.

3. Perhaps we will simplify the filter be removing very small coefficients, for speed of computation.

How do we find the time domain filter, x ,(or a), having FFT, X, (or A)? In the past we have found the time function that transforms to a certain FFT by solving

$$x_j = \sum_{n=0}^{N/2} \alpha_n \cos(2\pi n j/N + \phi_n)$$

where the amplitudes α_n and the phases ϕ_n are taken from the FFT, X where $x \overset{\text{FFT}}{\longleftrightarrow} X$. While we initially emphasized this method due to the way it clearly explains the FFT, it is simpler to just "invert" the FFT. In R, we perform this inversion by

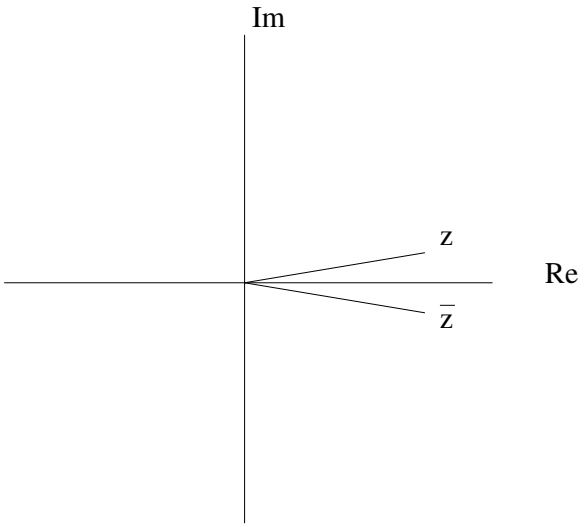
```
> x = fft(X,inverse=T) / N
```

Thus we simply need to invert our desired filter response to get our filter coefficients.

There is one particular issue that must be clarified first, however. While we know that the FFT of a N-point vector is also an N-point complex vector, we have, until now, only used the first half of the FFT. The reason for this is that if x is a real-valued vector (like any sound function), then the FFT of x, X, will have a "complex conjugate symmetry." This means that

$$X(N-n) = \bar{X}_(n)$$

where by $\bar{X}(n)$ we mean the complex conjugate of $X(n)$, which is the reflection of $X(n)$ across the real axis. Thus if $z = a + bi$, then $\bar{z} = a - bi$, as in the figure below.



Thus the 2nd half of the frequency does not contain any new information, justifying our lack of interest in it.

For the present situation, we want to take the inverse Fourier transform of our desired filter response, which we have specified for frequencies from 0 to $N/2$. To construct the remaining part of the filter response for frequencies $N/2+1$ to $N-1$, we simply fill in the values using the conjugate symmetry relation. In many cases, such as all three types of filters we have mentioned, our frequency response is *real* so we don't need to worry about the complex conjugates, and need only enforce the symmetry relation. That is, if our frequency response is given by A, then

$$A(N-n) = A(n)$$

or, if we are using 1-based indexing, as in R, then

$$A(N+2-n) = A(n)$$

Filling in missing part of the frequency response this way, we can simply invert to get the desired filter.

The program **lowpass_filter.r** demonstrates this technique and applies it to real audio samples.

Inverse of FFT

To be more formal, if x is an N-vector with a $x \overset{\text{FFT}}{\longleftrightarrow} X$ then, by definition we have

$$X(n) = \sum_{j=0}^{N-1} x_j e^{-2\pi i n j/N}$$

for $n = 0, \ldots, N-1$. It turns out to be a simple matter to invert this transform. A relatively simple calculation shows that x can be recovered by

$$x_j = \frac{1}{N} \sum_{n=0}^{N-1} X(n) e^{2\pi i n j / N}$$

Note that the inverse transform is almost identical to the original FFT; the only differences are that we divide by N and use a positive sign in the exponential function.

The inverse equation follows, since

$$\begin{aligned}
\frac{1}{N} \sum_{n=0}^{N-1} X(n) e^{2\pi i n j / N} &= \frac{1}{N} \sum_{n=0}^{N-1} \sum_{j'=0}^{N-1} x_{j'} e^{-2\pi i n j' / N} e^{2\pi i n j / N} \\
&= \sum_{j'=0}^{N-1} x_{j'} \frac{1}{N} \sum_{n=0}^{N-1} e^{-2\pi i n (j'-j)/N} \\
&= \sum_{j'=0}^{N-1} x_{j'} \delta(j'-j) \\
&= x_j
\end{aligned}$$

where

$$\delta(k) = \begin{cases} 1 & k = 0 \\ 0 & \text{otherwise} \end{cases}$$

Let's come back now to our original description of the FFT, where we claimed that if $x \overset{\text{FFT}}{\longleftrightarrow} X$, then

$$x_j = \sum_{n=0}^{N/2} \alpha_n \cos(2\pi n j / N + \phi_n) \tag{2.4}$$

holds, where the α's and ϕ's are given by

$$\alpha_n = \begin{cases} 2|X(n)|/N & n = 1 \ldots, N/2 - 1 \\ |X(n)|/N & n = 0, N/2 \end{cases}$$

and

$$\phi_n = \text{Arg}(X(n))$$

Assuming that x is *real*, we have complex conjugate symmetry in X, so that, using $z + \bar{z} = 2|z| \cos(\text{Arg}(z))$

$$\begin{aligned}
x_j &= \frac{1}{N} \sum_{n=0}^{N-1} X(n) e^{2\pi i n j / N} \\
&= \frac{1}{N} [X(0) + X(N/2)(-1)^j + \sum_{n=1}^{N/2-1} X(n) e^{2\pi i n j / N} + X(N-n) e^{2\pi i (N-n) j / N}] \\
&= \frac{1}{N} [X(0) + X(N/2)(-1)^j + \sum_{n=1}^{N/2-1} 2|X(n) e^{2\pi i n j / N}| \cos(\text{Arg}(X(n) e^{2\pi i n j / N}))] \\
&= \frac{1}{N} [X(0) + X(N/2)(-1)^j + \sum_{n=1}^{N/2-1} 2|X(n)| \cos(2\pi n j / N + \text{Arg}(X(n)))] \\
&= \sum_{n=0}^{N/2} \alpha_n \cos(2\pi n j / N + \phi_n)
\end{aligned}$$

where that last equation uses the fact that $X(0)$ and $X(N/2)$ are real.

Autoregression (ar_sine.r, ar.r)

We have seen that we can modify sounds in a wide variety of ways by filtering

$$\underbrace{y}_{\text{modified sound}} = \underbrace{a}_{\text{filter}} * \underbrace{x}_{\text{original sound}}$$

We saw that the result of filtering, in frequency domain, was to multiply the transform of the original sound by the transform of the filter. Thus some frequencies are suppressed while others are enhanced. A related idea is that of *autoregression* (AR).

A sequence is *autoregressive* if each value of the sequence is computed as a linear combination of its L predecessors:

$$y_j = a_1 y_{j-1} + a_2 y_{j-2} + \ldots + a_L y_{j-L} \tag{2.5}$$

The term autoregression comes from the way the signal is "regressed" (linearly modeled) on itself.

What kinds of sequences can be constructed this way? An interesting example is the 2nd order (using two predecessors) autoregressive (AR) sequence

$$y_j = 2\cos(\theta) y_{j-1} - y_{j-2}$$

We will not prove this, though it is easy to verify that the solution to this sequence is given by

$$y_j = a\sin(\theta j + \phi)$$

This is true no matter what the *initial conditions* of the sequence are — that is, no matter how we set y_0 and y_1. In this case we have no time units associated with the sample points, so it only makes sense to express the frequency in terms of phase advance per sample. Phase advance per sample is known as *angular frequency*, so the above sine wave has angular frequency θ. This 2nd order model is demonstrated in the program **ar_sine.r**.

The program also shows that a slight modification to the above equation:

$$y_j = 2r\cos(\theta) y_{j-1} - r^2 y_{j-2}$$

gives sequences of the form

$$y_j = ar^j \sin(\theta j + \phi)$$

These are exponentially decaying sines at angular frequency θ when $r < 1$ and exponentially increasing sines when $r > 1$. One can show that a pure AR model like Eqn. 2.5 will give a signal that is a sum of exponentially decaying or increasing sine waves.

A common variation on this model is to suppose that we don't have perfect prediction or regression and say the sequence can only be *approximated* by a linear combination of the last several values. That is

$$y_j = a_1 y_{j-1} + a_2 y_{j-2} + \ldots + a_L y_{j-L} + \epsilon_j$$

where $\epsilon_0, \epsilon_1, \epsilon_2, \ldots$ is a a sequence of 0-mean white noise. Thus the model assumes that the approximation is correct on average, since the noise is 0-mean. Such a model is said to be *driven* by the noise sequence ϵ.

An interesting variation on this is to assume that autoregression is driven by a known sequence, x:

$$y_j = a_1 y_{j-1} + a_2 y_{j-2} + \ldots + a_L y_{j-L} + x_j$$

Thus, $x_0, x_1, x_2, \ldots$ is our *input* sequence (think of our original audio) and $y_0, y_1, y_2, \ldots$ is the resulting *output* or modified sequence. We can write this by letting $a_0 = 1$, giving

$$a_0 y_j - a_1 y_{j-1} - a_2 y_{j-2} - \ldots - a_L y_{j-L} = x_j$$

or equivalently

$$a * y = x$$

where $a = (1, -a_1, -a_2, \ldots, -a_L)$. Taking the FFT of both sides gives $A \cdot Y = X$ and thus

$$Y = \frac{X}{A}$$

Note the similarity with our original filtering problem. With our first treatment of filtering the resulting sequence had transform that was the original transform *times* the filter transform. Here we *divide* by the filter transform. This is really considerably different from before. If we can find a filter whose transform is near 0 for some frequency θ, the effect of using the filter as an autoregressive filter will be to greatly amplify frequencies near θ. This is demonstrated in the program **ar.r**.

Chapter 3

Time-Frequency Sound Representations

Spectrogram (spectrogram.r)

Suppose we have a single instrument that plays three different pitches in succession. In the first rendition, y, the three notes are played in ascending order, while in a second rendition, z they are played in descending order. Since the frequency content of y and z are similar, their Fourier transforms will also be similar (at least in modulus). But they *sound* completely different. In this way, the Fourier transform fails us, by failing to represent the way frequency content evolves over time.

The spectrogram is an image that plots sound energy as a function of frequency (on the vertical axis) and time (on the horizontal axis), as in the figure below. Note that while each frequency bin represents a *complex* number, we are simply using the modulus (amplitude) of the number in creating our spectrogram image as in Figure 3.1.

Some important parameters of the spectrogram are the following:

1. Usually window before taking Fourier transforms (choice of window).

2. $N = \#$ points per FFT.

3. It is common to overlap frames. While this is not shown in Figure 3.1 it is demonstrated in the following figure. The amount of overlap is another important parameter.

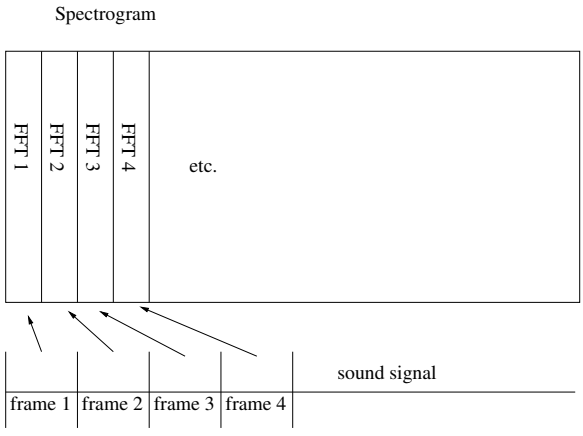
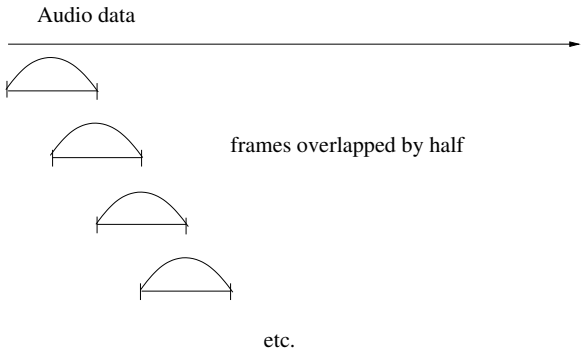


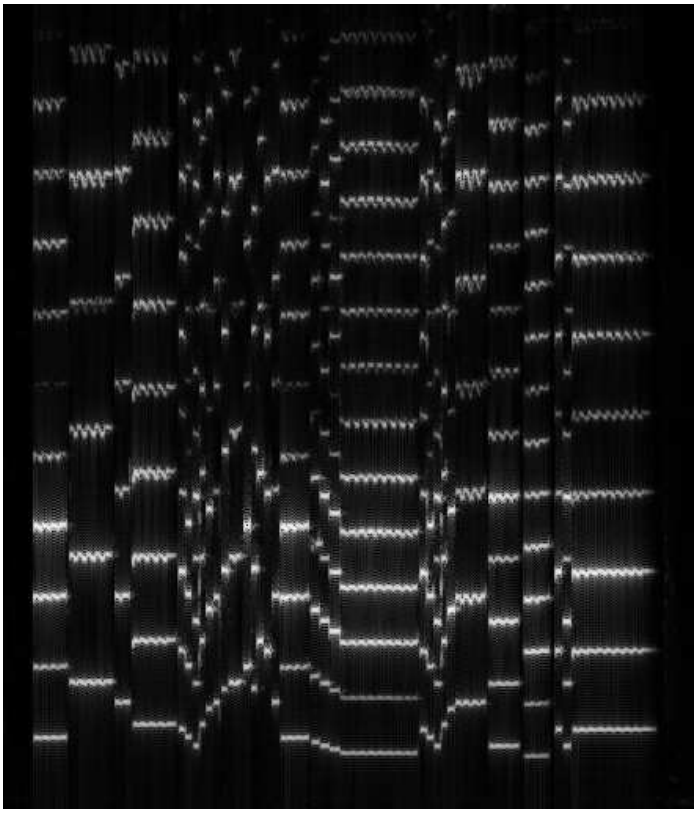
Figure 3.1: The construction of a spectrogram



Of the parameters, the most important is N, the number of points per FFT. As N decreases we have shorter frames, therefore more frames/sec and better time resolution. However we also have fewer points per FFT, fewer measured frequencies, worse frequency resolution. Luckily, in many cases it is possible to get good enough resolution in both time and frequency simultaneously. The program **spectrogram.r** demonstrates this trade off.

The spectrogram is a useful tool for looking at sound. While any visualization of sound is necessarily an abstraction, our powers of visual analysis are, in some ways, more flexible than for audio. For the most part, an audio signal must be understand as it happens, since we have only limited ability to analyze sound that occurred in the past. In contrast, we choose the order, rate, and manner in which we visit an image. Since the image does not need to be digested in real-time, we sometimes see things that are harder to hear.

When viewing a spectrogram of pitched sound, the harmonic structure of the individual notes is clearly seen as a collection of evenly spaced horizontal "lines" for each note. As with a single spectrum, the pitch that we hear will be the frequency associated with the lowest harmonic, which is the same as the spacing between the harmonics. Horizontal (one note after the other) intervals are also easy to recognize from a spectrogram. Suppose we have an interval for which we know the approximate ratio of frequencies, such as 3:2 for a perfect fifth. If f_1 is the frequency of the lower pitch and f_2 is that of the higher pitch, then we know that, approximately, $3 \times f_1 = 2 \times f_2$. In other words, the 3rd harmonic of the lower pitch should be at the same vertical position as the 2nd harmonic of the upper pitch — a so-called "shared harmonic." More generally we see a shared harmonic between two pitches, say the mth harmonic of one pitch equaling the nth harmonic of another, then the ratio of frequencies is m/n. We can recognize many such intervals if we can identify the corresponding simple ratios.



For instance, in the above figure the 2nd and 4th notes clearly have a 3:2 ratio so we see that the 2nd note is a perfect 5th higher than the 4th note. Similarly the 4th harmonic of the 3rd note is close to the 5th harmonic of the 4th note, thus these notes have the frequency relationship of 5:4 and constitute a major third.

We can also recognize the traits of noisy sounds from a spectrogram in terms of high and low frequency content and, perhaps more important, the time evolution of these traits. The **spectrogram.r** program can be used to examine the variety of percussion sounds on the course web page and, perhaps, even to identify them. We will discuss this in detail in class.

Time-Frequency Sound Representations

The spectrogram is created by plotting just the amplitude or modulus of each time-frequency bin as brightness. While this corresponds quite naturally to much of what we hear in sound — time evolving frequency energy — the spectrogram discards the important information of the phase. In fact, there may be many time functions that would produce the same, or nearly the same, spectrogram, so it is not possible to recover the time domain signal from

a spectrogram. This is unfortunate since, though the spectrogram domain seems like an natural place to perform modifications of our sound, we have no way to convert these processed images back into sound. This can be remedied with the time-frequency sound representation we discuss here.

The *short time Fourier transform* (STFT) is analogous to the spectrogram, except that we retain a *complex number* for each time-frequency bin. Thus the picture of the STFT is identical to Figure 3.1, except we do not discard the phase information.

Stated informally, let $w(j)$ be our N-point window function where we regard $w(j) = 0$ for $j \notin \{0, \ldots, N-1\}$. Let $x_0, x_1, \ldots$ be our signal. Let H be the *hop size* (the number of samples between STFT "frames.") If X is the STFT of x, we will write

$$x \overset{\text{STFT}}{\longleftrightarrow} X$$

where $X = X(t, n)$ is given by $X(t, \cdot) = FFT(w \times t\text{th frame of } x)$ — that is, the tth column of X is the FFT of the windowed tth frame of the signal.

More formally,

$$X(t, n) = \sum_{j=0}^{N-1} x(tH + j)w(j)e^{\frac{-2\pi ijn}{N}}$$

for $t = 0, 1, \ldots$ and $k = 0, 1, \ldots, N-1$.

How do we recover our original sound signal x from the STFT X? Suppose that our window function, w, has the property that the sum of all hop-sized translates of the window gives the constant function 1. For instance, consider the triangle window in the figure below with maximum value of 1, and a hop size of half the window length. At the peak of a triangle, there is only one window function contributing to the sum — a single window at its maximum, so the sum is, of course, 1 at that point. As we move forward from the window peak, the current window begins to decrease while a new window begins to enter the picture. However, the rate of increase of the incoming window is exactly the rate of decrease of the outgoing window. Thus, the sum remains 1 as we move away from the window peak. This argument continues until we arrive at the next window peak, at which point we employ the same argument with the following two windows. Thus we have

$$\sum_t w(j - tH) = 1$$

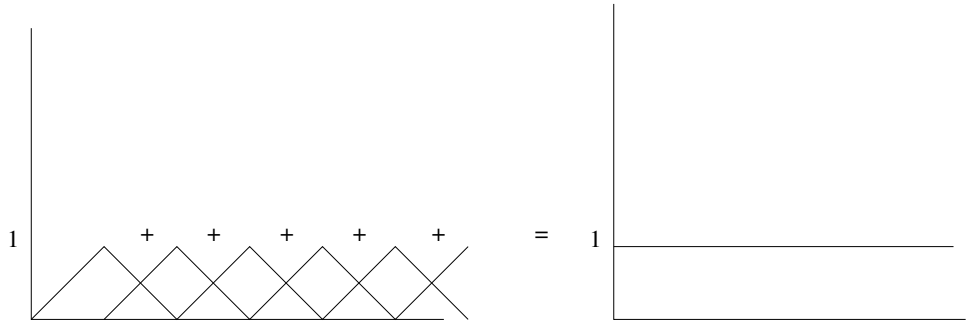
for all j. (Recall that we define the window function for all j, though it is 0 for $j \notin \{0, \ldots, N-1\}$). In terms of the picture, it may be helpful to think of this equation as

$$\sum_t w_{tH} = 1$$

where the vector w_{tH} is defined for all j by

$$w_{tH}(j) = w(j - tH)$$

This notation means that we think of each w_{tH} as a function or vector. This function is the window shifted so that it begins at tH.



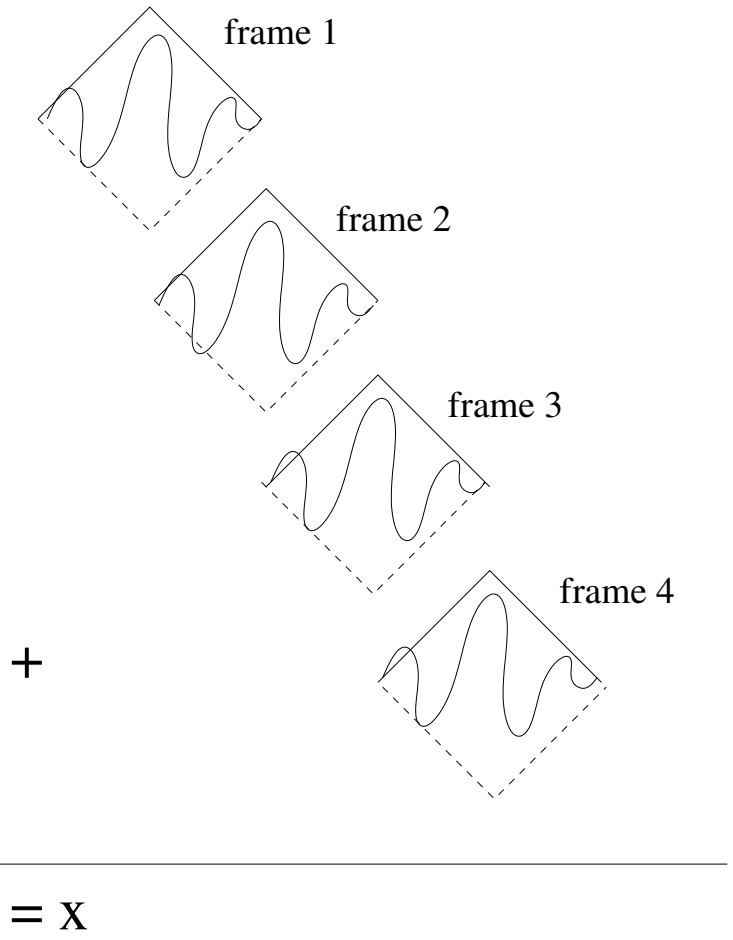
Thus, if we add up all of the windowed audio frames we recover the original signal:

$$\begin{aligned} x_j &= x_j \cdot 1 \\ &= x_j \sum_t w(j - tH) \\ &= \sum_t x_j w(j - tH) \end{aligned}$$

Using the vector notation, this says that

$$x = \sum_t x w_{tH} \tag{3.1}$$

as illustrated in the following figure:



Eqn. 3.1 says that our signal can be constructed by "fading in" and "fading out" between a collection of "grains" — little pieces of sound. This fade-in-fade-out technique makes sense no matter where the grains of sound come from. Thus rather interesting collages of sound can be assembled through

$$x = \sum_t x_{tH} w_{tH}$$

where the grains, x_{tH} are whatever we like. This technique is known as "granular synthesis."

Inverting the STFT (stft.r)

Suppose that

$$x \overset{\text{STFT}}{\longleftrightarrow} X$$

and that the STFT is constructed using the window length N, hop size H, and window function w having the property

$$\sum_t w^2(j - tH) = 1$$

for all j. Equivalently, we could write this as

$$\sum_t w_{tH}^2 = 1$$

where we mean the constant vector 1 on the right. For example, consider our favorite (and only) Hann window

$$w(j) = \begin{cases} \frac{1+\cos(2\pi j/N-\pi)}{2\sqrt{3/2}} & j = 0, 1, \ldots, N-1 \\ 0 & \text{otherwise} \end{cases}$$

where we have added a constant of $\sqrt{3/2}$ in the denominator. The program **hanning_translate.r** demonstrates that the Hann window has this property when $H = N/4$, though it is easy to verify by direct computation using basic properties of sine and cosine. By the definition of the STFT, we know that

$$\text{FFT}^{-1}(X(t, \cdot)) = (t\text{th frame of } x)w$$

so that

$$\text{FFT}^{-1}(X(t, \cdot))w = (t\text{th frame of } x)w^2$$

From this it follows that

$$\begin{aligned} x_j &= x_j \sum_t w^2(j - tH) \\ &= \sum_t \underbrace{x_j w(j - tH)}_{\text{inverse of } X(t,\cdot) \text{ shifted to } tH} w(j - tH) \\ &= \sum_t \frac{1}{N} \sum_{n=0}^{N-1} X(t, n) e^{\frac{2\pi i(j-tH)n}{N}} w(j - tH) \end{aligned}$$

This formula simply says that, to recover x, we simply take the inverse FFT's of each column of X, multiply these again by our window function, and add these up, treating the tth inverse FFT as a N-length sequence beginning at tH.

Writing this in our vector notation, this reads

$$x = \sum_t \frac{1}{N} (\mathbf{FFT}^{-1} X(t, \cdot))_{tH} w_{tH}$$

where, in both cases, the tH subscript means shifting by tH. Thus we simply overlap and add the shifted and windowed inverse Fourier transforms.

This process is demonstrated in the R program **stft.r**. In this program we define two functions, one for the STFT and one for the inverse STFT. The program demonstrates that we can recover the original time sequence by play the audio that is recovered using the inversion formula.

STFT Applications 1: Compression (compression.r)

The STFT gives a representation of an audio signal, x, as

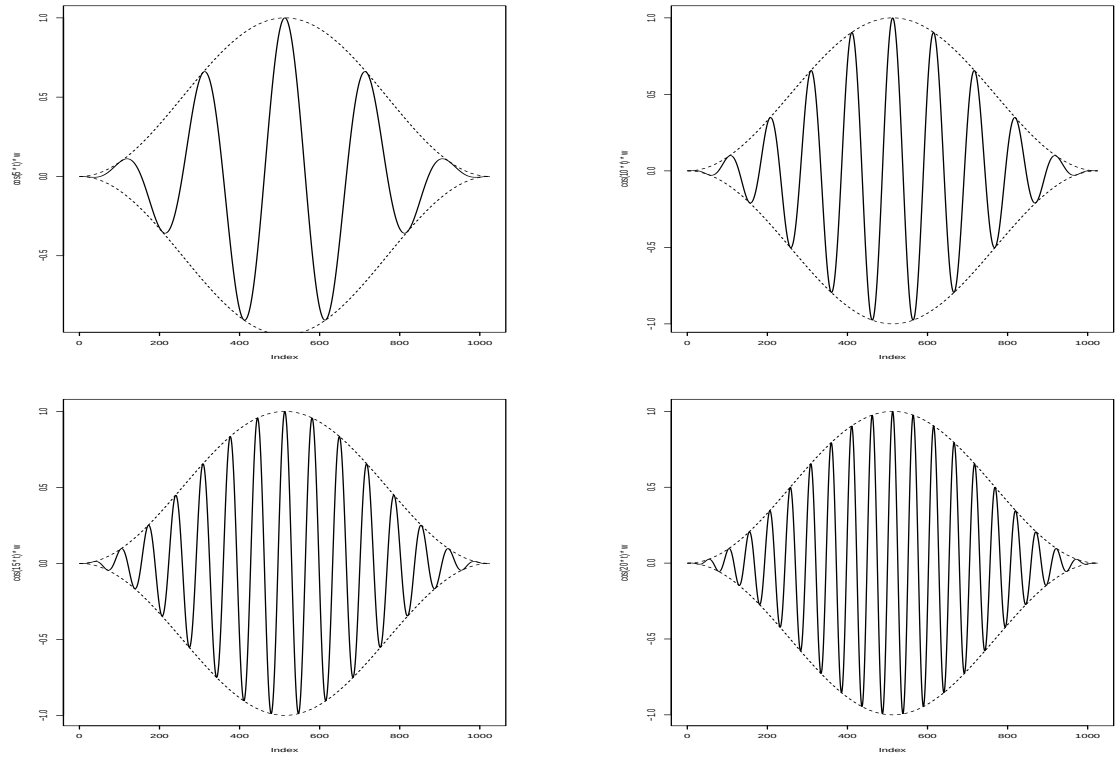
$$x_j = \sum_{t,n} \frac{1}{N} X(t, n) e^{2\pi i n(j-tH)/N} w(j - tH)$$

through our inversion formula. Writing $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, we can express x as

$$\begin{aligned} x(j) &= \sum_{t,n} \underbrace{\frac{1}{N}\mathrm{Re}(X(t,n))}_{\alpha_{tn}} \underbrace{\cos(2\pi n(j-tH)/N)w(j-tH)}_{\phi_{tn}(j)} + \underbrace{-\frac{1}{N}\mathrm{Im}(X(t,n))}_{\beta_{tn}} \underbrace{\sin(2\pi n(j-tH)/N)w(j-tH)}_{\psi_{tn}(j)} \\ &= \sum_{t,n} \alpha_{tn}\phi_{tn}(j) + \beta_{tn}\psi_{tn}(j) \end{aligned}$$

Thus x is expanded in terms of the basis functions ϕ_{tn}, ψ_{tn}.

These basis functions are simply windowed sinusoids as in the figure below, occurring at various places in the time domain as given by t (the functions ϕ_{tn} and ψ_{tn} are shaped like those depicted, but are offset by tH.)



Such a representation has interesting applications in signal *denoising*. Viewed in terms of the representation above, many of the $\{\alpha_{tn}, \beta_{tn}\}$ coefficients of an audio file will be negligibly small. Setting such coefficients to zero will have the effect of *denoising* the signal since most of these coefficients may, in fact, be due to unwanted noise.

Similarly, the above basis representation can be used as a means for *audio compression*. We can approximate our signal by retaining only the largest p percent of the coefficients, and assuming the others are 0. Of course, our compressed audio file would need to give both the (t, n) locations we choose to retain, as well as the associated coefficients. The quality of the signal will degrade as p decreases to 0, but there may be usable audio representations along the way that require significantly less storage than the original audio representation. Both of these possibilities, denoising and compression, are demonstrated in the R program **compression.r**.

STFT Applications 2: Desoloing

Suppose we begin with a recording x of a soloist with accompaniment, such as a violin soloist with orchestra, or a vocal soloist with rock band. Such a signal could be represented as a sum of these two parts:

$$x = s + o$$

where x, s, o are audio vectors with s and o the solo and accompaniment parts of the "signal." (Remember that the result of these two sources is *additive*). We would like to decompose x into s and o, thus recovering an accompaniment,

o that could be used for Karaoke or other purposes, and recovering a recording of the soloist s in isolation. How can we do this?

An equivalent problem statement is to decompose the STFT, X, as

$$X(t,n) = S(t,n) + O(t,n)$$

where $S(t,n)$ and $O(t,n)$ are the solo and accompaniment STFTs. This problem (in either domain) is known as *blind source separation* and is quite difficult.

One possible approach is to classify each time-frequency point (t,n) as either solo or accompaniment. We could represent this classification by a function

$$1_S(t,n) = \begin{cases} 1 & \text{if } (t,n) \text{ classified as solo} \\ 0 & \text{otherwise} \end{cases}$$

where S is the set of (t,n) points classified as solo. We then could approximate our solo and orchestra decomposition by

$$\begin{aligned} \hat{S} &= 1_S X \\ \hat{O} &= 1_{S^c} X \end{aligned}$$

where S^c is the complement (everything else) of S Such an approach is known as *masking* where 1_S is the *mask* that is applied to recover the solo part.

In this case it is clear that we have

$$X = \hat{S} + \hat{O}$$

essentially by definition. We can recover our approximations of solo and accompaniment by

$$\begin{aligned} \hat{s} &\overset{\text{STFT}}{\longleftrightarrow} \hat{S} \\ \hat{o} &\overset{\text{STFT}}{\longleftrightarrow} \hat{O} \end{aligned}$$

Thus this approach phrases source separation as a binary classification problem where we classify each (t,n) point as solo or accompaniment. Such an approach may work reasonably well when the majority of (t,n) are primarily the result of *either* the solo *or* the accompaniment. Time-frequency points where both sources make significant contributions are known as *collision* points. In general, it is very difficult to separate such points into their two contributions.

An alternative to the *blind* (no prior knowledge) approach uses a score-match between the audio data and the symbolic representation of the music — this is an alignment between a score and an audio file indicating where in the audio file the various score events occur. We will discuss the problem of estimating such a correspondence later in this class. With such a representation, we know the times of each of the solo notes, as well as the approximate fundamental frequencies for these notes, as specified in the score. Thus we know the approximate (t,n) time-frequency points that correspond to the solo. In performing source separation we could either work directly from the score match to estimate S, or try to combine both the score match and other attributes of the audio data itself to estimate S. This latter approach is demonstrated at **http//:xavier.informatics.indiana.edu/~craphael/cmj07** on both artificially mixed, and real recordings.

The Importance of Phase (phase_scramble.r, robot_whisper.r)

We have claimed that some situations allow us to vary phase in our sound representation with little or no perceptual change. Such an example is the additive synthesis model for a pitched sound

$$\begin{aligned} x(t) &= \alpha_1 \sin(2\pi f t + \phi_1) + \alpha_2 \sin(2\pi 2 f t + \phi_2) + \ldots + \alpha_n \sin(2\pi n f t + \phi_n) \\ &= \sum_{k=1}^{n} \alpha_k \sin(2\pi k f t + \phi_k) \end{aligned}$$

In this model we get no perceivable change in sound as we choose different values for the *phases* $\{\phi_k\}$. This is demonstrated by the **phase_scramble.r** program, which repeatedly uses the additive synthesis model with the same

amplitudes, but randomly chosen phases. The same is true of the extension of this idea to a sound with time-varying pitch, $f(t)$:

$$x(t) = \sum_{k=1}^{n} \alpha_k \sin(2\pi k \int_0^t f(\tau)d\tau + \phi_k).$$

In contrast, the choice of the phase in our STFT time-frequency representation, plays a crucial role in determining the nature of the corresponding sound. One very simple example is when we take an STFT of actual sound, set all the phases to 0, and invert the STFT. That is

$$\begin{array}{rcl} x & \overset{\text{STFT}}{\longleftrightarrow} & X \\ \hat{X}(t,n) & = & |X(t,n)| + 0i \\ \hat{x} & \overset{\text{STFT}}{\longleftrightarrow} & \hat{X} \end{array}$$

The original and resulting audio can be heard through the program **robot_whisper.r**. While the original audio may have a variety of time-varying pitch content, the "roboticized" audio comes across at a single pitch. This is something like the difference between regular speech and monotone speech. Why does this happen?

Let us first suppose that our sound is pitched (perhaps with several pitches) and completely steady state, not changing in pitch, amplitude, or timbre. In this case the spectrogram of the sound would appear as a collection of completely straight horizontal lines corresponding to the frequencies present in the signal. Since the frequency content is constant over our frames, and we have set all phases to 0, the "columns" of our STFT, $\hat{X}(t,\cdot)$, for $t = 0, 1, \ldots$ are identical. Suppose we let

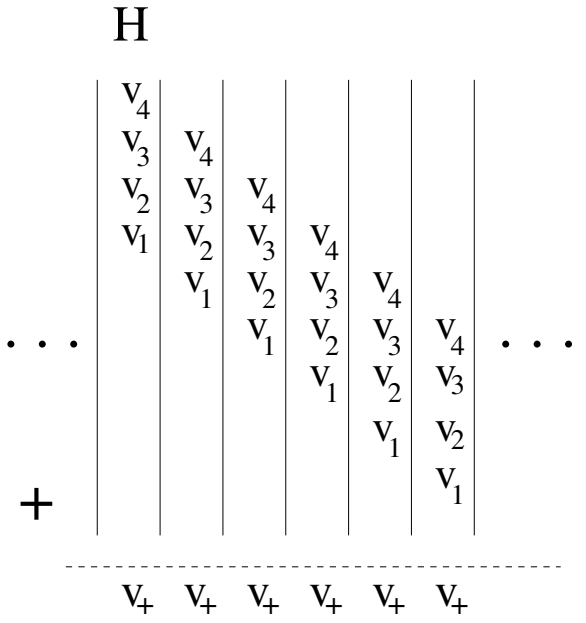
$$v = w \cdot \text{FFT}^{-1}\hat{X}(t,\cdot)$$

be the inverse FFT of one of these columns (they are all the same), multiplied by our window function.

Our STFT inversion algorithm tells us we must reconstruct the time signal by adding together shifted copies of v:

$$\hat{x} = \sum_t v_{tH}$$

where, as usual, $v_{tH}(j) = v(j - tH)$. Suppose, to be definite, $H = N/4$ and let v_1, v_2, v_3, v_4 be the frame v divided into 4 equal pieces of length H. Now when we overlap add, the process looks as depicted below:



Note that, after adding, we make an identical contribution of $v_+ = v_1 + v_2 + v_3 + v_4$ to each frame. Thus the signal is periodic with period H

In a more realistic situation the sound is only approximately steady-state, though the steady-state approximation is pretty good for many real-world sounds when H corresponds to a short time interval. In this case our reconstructed signal, $\hat{x}$ is approximately periodic with period H, though the sense of fixed pitch is still retained.

Relating this to what we hear, the audio example in **robot_whisper.r** uses a sample rate of SR = 48kHz, with frames of length $N = 1024$ and a hop size of $H = N/4$. Thus we hear the frequency of associated with period H:

$$\frac{\text{SR}}{\text{H}} = \frac{48000}{1024/4} = 187.5Hz.$$

One can check that this is, in fact, the frequency we hear for the program.

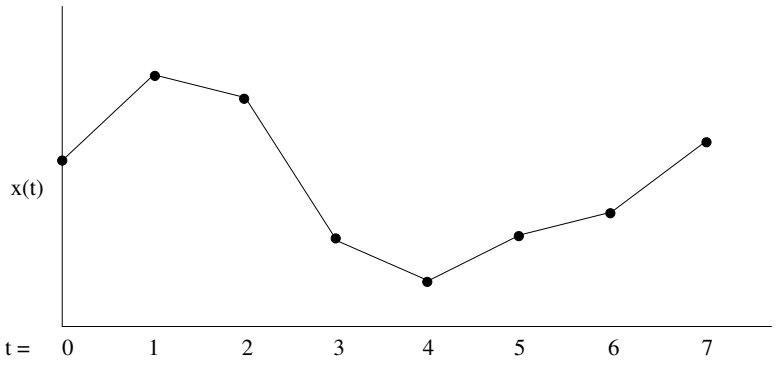
A similar idea, rather than setting the phases to 0, *randomizes* the phases. This produces the effect of making the audio sound as if it is whispered — that is, taking away all sense of pitch to the sound and leaving only the "noisy" component. This is also demonstrated in the **robot_whisper.r** program.

Resampling (resampling.r)

Suppose we have music audio that is played at a tempo of 60 beats per minute. We would like to transform this audio so it is played at a rate of 90 beats per minute. How can we do this? The simplest possible approach is to *resample* the audio. While there are many possible ways to implement resampling, we present only the simplest of these here.

Imagine that our audio samples are $x_0, x_1, \ldots, x_T$. Since we want to play the audio 3/2 times as fast, we need to create a total of 2 samples of the new audio for every 3 samples of the original audio. This will involve estimating the value of the signal *between* audio samples.

The *linear interpolation* approach, simply imagines that we create a continuous sound signal $x(t)$ for *all* possible times t (not just at the sample values of $t = 0, 1, 2, \ldots$. Pictorially, we simply connect the samples values by lines segments, as below.



More exactly, if $t > 0$ is an arbitrary time point, in units of samples, we interpolate $x(t)$ as

$$\begin{aligned} x(t) &= px_{\lfloor t \rfloor} + qx_{\lceil t \rceil} \\ &= (\lceil t \rceil - t)x_{\lfloor t \rfloor} + (t - \lfloor t \rfloor)x_{\lceil t \rceil} \end{aligned}$$

where $\lfloor t \rfloor$ and $\lceil t \rceil$ are the integers below and above t.

Good News: As advertised, this has the effect of creating the desired tempo change in the audio.

Bad News: In addition, we create a pitch change. In particular for the case mentioned above we have moved the pitch up a perfect 5th.

These phenomena are demonstrated in the R program, **resampling.r**.

We see now that what seemed to be a simple notion — just changing the tempo of the audio — becomes somewhat murky from a scientific perspective. That is, we are trying to change the rate of the audio at the note level (tempo), while *not* changing the rate of audio at the period level (pitch). Thus the goal of audio time-stretching is perceptually clear, but not well-posed from a scientific standpoint. We will proceed bravely, without having clearly articulated our goal.

The Shannon Sampling Theorem

This section on the Shannon theorem constitutes a brief digression from the main thrust of our course. This theorem is one of the main results of any serious introductory class to signal processing, and it is too good to completely leave out. Though it might not have many direct applications for our purposes, other than resampling, it is a gem that is worth knowing.

Linear interpolation allows us to take a sequence $x_0, x_1, \ldots$ and to produce a function $x(t)$ that "agrees" with the sampled values, yet gives a value for all t. While linear interpolation methods may work reasonably well for a variety of situations, such as small changes of playback rate and pitch, such techniques can function quite poorly in other cases, such as resampling to lower sampling rate. A surprising fact is that there is a "correct" way to recover the continuous-time signal, $x(t)$, from its samples, known as the *Shannon Sampling Theorem*.

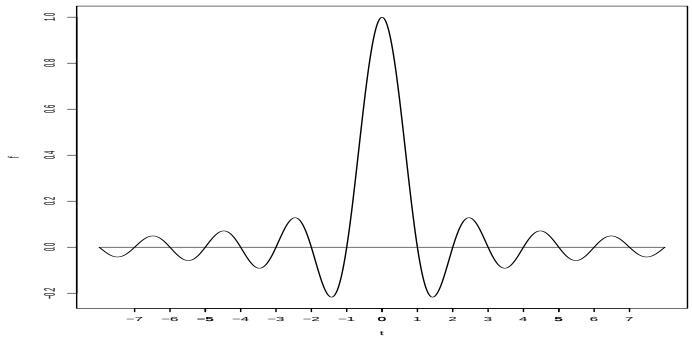
The hypothesis of the Shannon Theorem deals with a time signal, $x(t)$, *before* it has been sampled. Recall that if we have a sampling rate of SR, then the highest frequency we can represent is the so-called Nyquist frequency, SR/2. The Shannon theorem assumes that the original signal, $x(t)$, has no frequency content above SR/2. If it did, we know the sampling procedure would produce aliasing. Assuming this criterion is met, we can recover the original signal, $x(t)$, *exactly* only knowing the samples $\ldots, x_{-2}, x_{-1}, x_0, x_1, x_2, \ldots$. This is done according to the formula

$$x(t) = \sum_{j=-\infty}^{\infty} x_j \phi(t - j) \tag{3.2}$$

where $\phi(t)$ is known as the *sinc function*, and defined by

$$\phi(t) = \frac{\sin(\pi t)}{\pi t}$$

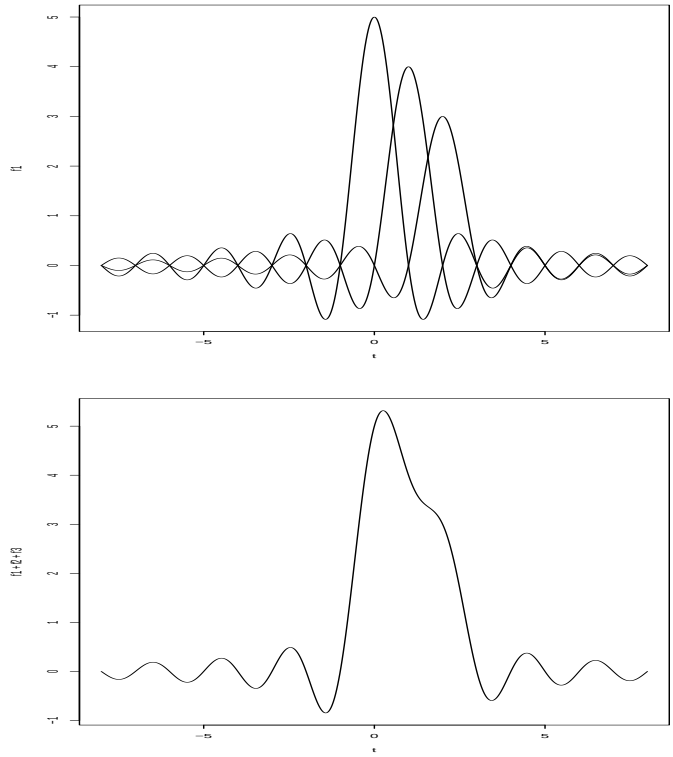
Though it may not be clear from the definition, the right way to define $\phi(0)$ is by $\phi(0) = \lim_{t \to 0} \phi(t) = 1$. The sinc function is shown in the figure below:



Note that $\phi(t)$ is 1 at 0 and is 0 at all of the other integral values of t, thus using Eqn. 3.2 for integral j' we must get

$$x(j') = \sum_{j=-\infty}^{\infty} x_j \phi(j' - j) = x_{j'}$$

That is, $x(t)$ is consistent with the sampled values, which is comforting. Pictorially, Eqn. 3.2 says that we reconstruct the original function $x(t)$ by adding up scaled sinc functions at the sample locations, where we use the sample values as the scaling constants, as shown below.



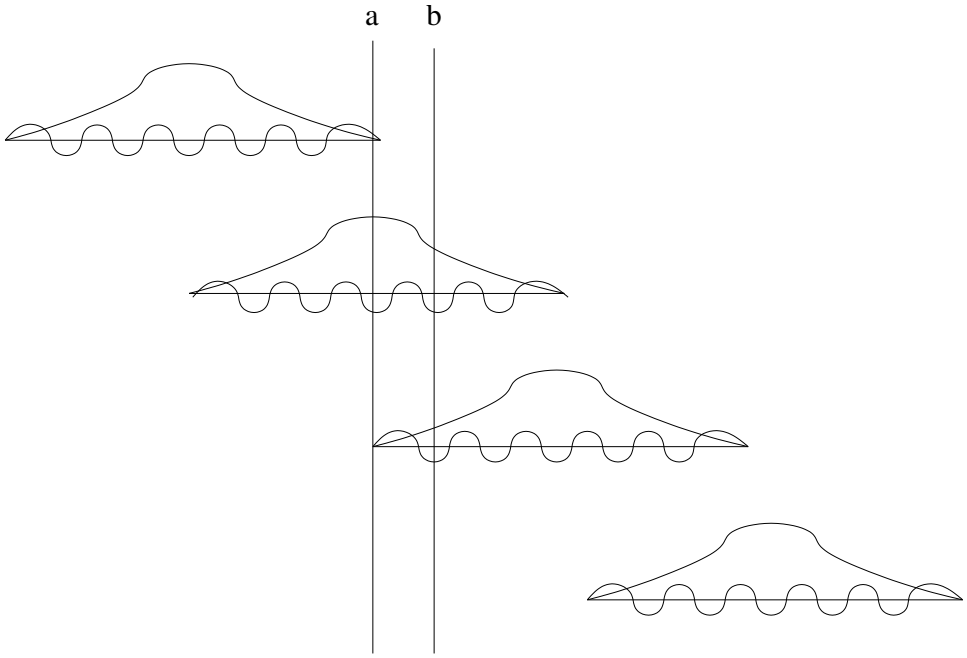
The upper figure shows the situation in which all of our samples are 0 except for the ones collected at 0, 1 and 2, where we have the sample values 5,4, and 3. We reconstruct by putting a sinc function at the locations 0,1 and 2, scaled by the sample values, as shown in the top panel of the figure. Our reconstruction is the sum of these functions, shown in the bottom panel.

The Shannon theorem gives a more principled alternative to resampling by linear interpolation. Linear interpolation makes the unrealistic assumption that the our time function continues linearly between sample values. The Shannon theorem tells us exactly how the unseen parts of the function behave, only by considering the sampled values. Better results can be achieved by using (or approximating) the continuous time values given by the Shannon theorem when resampling.

Phase Vocoder (phase_vocoder.r)

Consider the following experiment in which we try to make continuous sound out of a single FFT spectrum. To do this, we take our FFT and replicate it over and over to produce our STFT. We then use the usual inversion formula to construct the actual sound. This experiment is demonstrated in the R program **phase_vocoder_motivation.r**. Clearly it doesn't work, though it would be a good idea to try to understand why.

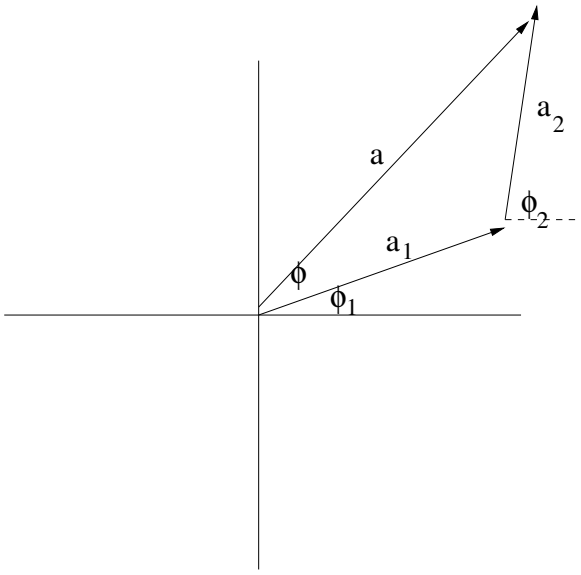
It is easiest to visualize the problem when our sound consists of a *single* sine wave, though not necessarily one of the FFT analysis frequencies. In particular, we assume that our sine wave is *not* periodic over H samples. In this case our reconstruction of the signal will be the sum of the windowed sinusoid, overlapped as in the picture below.



To understand the effect of the situation we make a brief digression. Suppose we have a sum of two sine waves each of the *same* frequency, but with different amplitudes and phases:

$$x(t) = a_1 \sin(2\pi f t + \phi_1) + a_2 \sin(2\pi f t + \phi_2)$$

It is well-known that the result is also a sine wave of frequency f, with an amplitude and phase having a simple relation to the a_1, a_2, ϕ_1, ϕ_2. The result can be visualized using the "parallelogram" rule. That is, if we add the vector with length a_1 and angle ϕ_1 to the vector with length a_2 and phase ϕ_2, we get a new vector whose length and angle are the amplitude and phase of the resulting sine wave. This is demonstrated in the picture below.

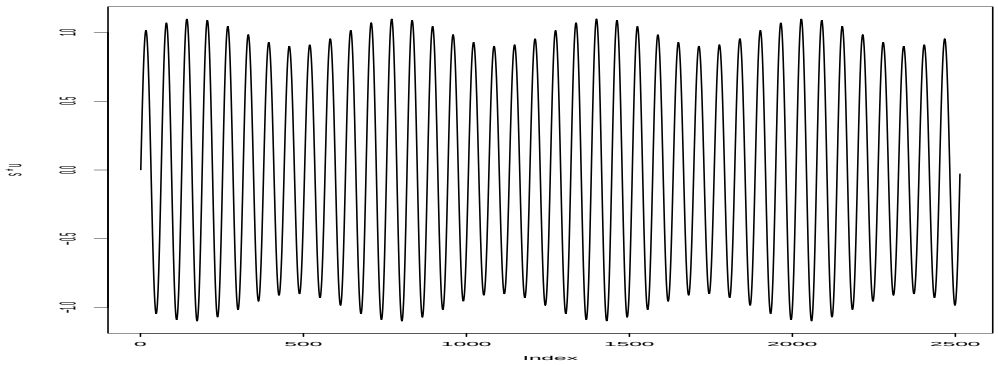


From the picture we see that when we add two sine waves that have identical phases, then the amplitudes add. Otherwise there is some degree of *cancellation* between the sines and the resulting amplitude will be less than the

sum of the amplitudes.

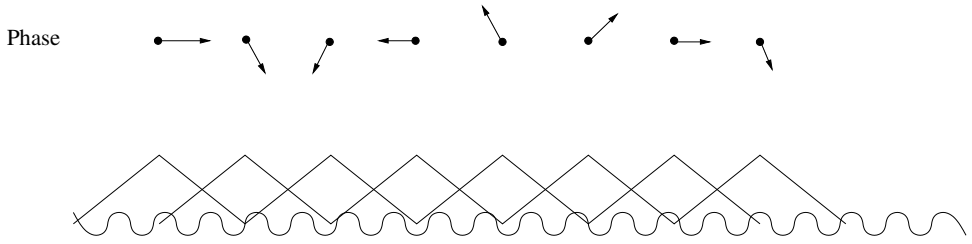
In terms of our original picture, consider the two time points a and b, indicated by the vertical lines. At a our overlap and add only involves a single sine wave, since we are at the center of the window and $H = N/2$. Thus in the neighborhood of a we would see a sine wave of the original amplitude. At the point b we are adding together two sine waves each scaled by $1/2$ due to the Hann window. Since our signal is not H-periodic, the two sines will have different phase. Thus, when we combine them at point b there will be some cancellation, resulting in a decrease in the resulting amplitude. Of course, the amount of interference we vary smoothly as we move between a and b.

If we look at the result from a more global perspective, we will see the resulting overlap add process oscillating between higher and lower resulting amplitudes as we move through the regions of various degrees of cancellation. The result will be a sine wave whose amplitude varies periodically with period given by the hop size H, as in the picture below. Clearly this is not what we want.



This is the picture of what we heard in the audio produced by **phase_vocoder_motivation.r**. Luckily, there is a simple way to fix this problem, as follows.

Suppose we compute the STFT for our original (non H-periodic) sinusoidal signal, $x(t)$. There is, of course, nothing interesting happening as far as the amplitude of the individuals FFTs are concerned, each showing a peak around the relevant frequency, f. However, we will see a different phase in each of the frames, as indicated below.



Moreover, since our frequency is constant, the difference in phase between adjacent frames will also be a constant. We'll call this phase difference $\Delta\phi$, measured in radians. Thus, if we wanted to avoid cancellation when our overlapped frames are added, we would offset each successive frame by $\Delta\phi$ over the previous frame. Thus, for our single sinusoid, the frames would be

$$\begin{aligned}
x_0(t) &= w(t)\sin(2\pi ft) \\
x_1(t) &= w(t-1H)\sin(2\pi ft + \Delta\phi - 1H) \\
x_2(t) &= w(t-2H)\sin(2\pi ft + 2\Delta\phi - 2H) \\
x_3(t) &= w(t-3H)\sin(2\pi ft + 3\Delta\phi - 3H) \\
\vdots &\quad \vdots
\end{aligned}$$

The idea of the phase vocoder is to use this same argument for each of the FFT frequencies, and for each windowed "grain" of sound. For the phase vocoder, we take a "snapshot" of sound for each overlapped frame as an FFT. However, rather than saving the actual phase for each analysis frequency, we save the *difference* in phase. We retain the original modulus for each FFT, exactly as before. Said more precisely, if

$$x \overset{\text{STFT}}{\longleftrightarrow} X$$

defined as before, we construct the modified STFT, $\bar{X}$, by

$$\bar{X}(t,n) = |X(t,n)|e^{i\Delta\phi(t,n)}$$

where

$$\Delta\phi(t,n) = \begin{cases} \text{Arg}(X(t,n)) & \text{if } t = 0 \\ \text{Arg}(X(t,n)) - \text{Arg}(X(t-1,n)) & \text{otherwise} \end{cases}$$

That is, for each row of the STFT we construct our modified STFT simply by substituting phase differences for actual phase.

It is a simple matter to invert $\bar{X}$. All we need to do is to convert the modified STFT, $\bar{X}$, to the regular STFT, X, by *undoing* the differencing. That is, if $\phi_0, \phi_1, \phi_2, \ldots$ are the sequence of phases we encounter on a particular row of the X, then the corresponding row of $\bar{X}$ will have $\phi_0, \phi_1 - \phi_0, \phi_2 - \phi_1, \phi_3 - \phi_2, \ldots$. We reconstruct the original phase simply by "integrating" (performing the **cumsum**) on this sequence. This gives

$$\begin{aligned} \phi_0 &= \phi_0 \\ \phi_1 &= \phi_0 + (\phi_1 - \phi_0) \\ \phi_2 &= \phi_0 + (\phi_1 - \phi_0) + (\phi_2 - \phi_1) \\ \phi_3 &= \phi_0 + (\phi_1 - \phi_0) + (\phi_2 - \phi_1) + (\phi_3 - \phi_2) \\ \vdots &\quad \vdots \end{aligned}$$

Using this idea we rewrite our STFT routine to compute the modified STFT, $\bar{X}$, instead of the straight STFT, X. To recover audio from $\bar{X}$, we simply need to perform **cumsum** on each of the rows of $\bar{X}$ and then continue with the regular STFT inversion. This modified STFT and its inverse are given as functions in the R program **phase_vocoder.r**.

Why would we want to add this extra and, perhaps, seemingly pointless step to our definition of the STFT? Returning to our original goal of creating audio by replicating a single frame over and over, suppose we replicate over and over the first column of $\bar{X}$ rather than the first column of X. Once we do this we no longer have problem with changing amounts of cancellation between our audio frames during the inversion process. More generally, the modified view of the STFT guarantees that, for each analysis frequency, each new frame continues the phase from where the previous frame left off. Thus, we can now construct an STFT by piecing together STFT frames in any manner we like.

Phase Vocoder Examples

The program **phase_vocoder.r** shows two simple examples in which we perform:

time stretching by either replicating or deleting some of the STFT frames

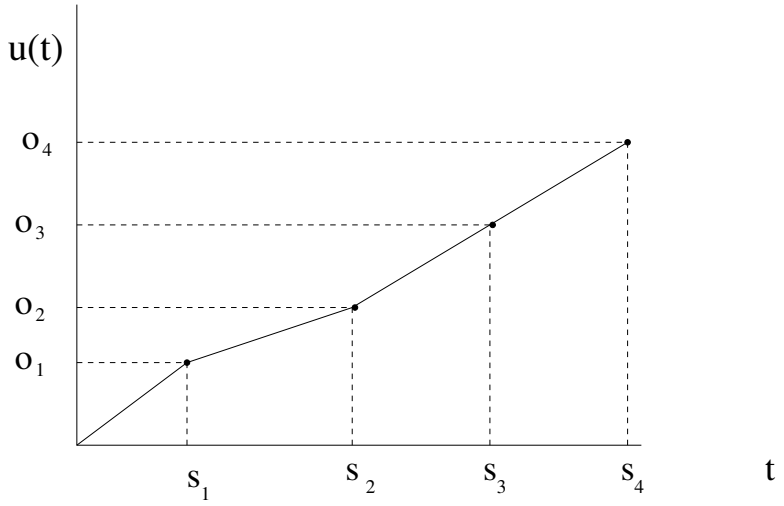
random assembly since the phase vocoder is designed to create phase discontinuity, even randomizing the order of STFT frames will produce continuous sounding audio.

In addition we will do two examples in class that are not computed in R, and do not have corresponding programs on the class website. The first of these shows a *real-time* implementation of phase vocoding that allows a person to play through an audio file, continually changing the play rate. When the play rate is 1, we get the original audio, nearly exactly as it was, courtesy of the STFT inversion formula. As the rate changes, we assemble frames by calculating our current position, in frames, according to the play rate. For instance, if the play rate is .5, we would advance 1/2 a frame for each output frame. That is, we would repeat each frame twice. Using this idea it is possible to move through the audio at any rate, including 0 (stalling the sound at a particular point), or a negative rate

(moving backwards through the audio file). It is interesting to note that, while we retain phase continuity through our phase vocoding procedure, we introduce other undesirable elements into the sound. For instance as we slow down the playback rate, we also slow down the reverberation decay of the room, the attack time, the vibrato rate, and other musical attributes that normally do not depend on tempo. These can produce unrealistic sounding audio, though are often not noticeable at play rate near 1.

A second example we will show in class is called the "Oracle." With the Oracle, we take a recording of a solo instrument playing in isolation. Using score following, as will be discussed, we can approximate the audio positions of all of the notes in the soloist's score. We will do the same for a recording of the orchestral accompaniment to the soloist. Even though these two performances were made with no knowledge whatsoever of each other, we can use phase vocoding to match the timing of the orchestra performance to to soloist, thereby generating an orchestral accompaniment *after the fact*.

To do this, suppose that k indexes the score positions at which we have have coincident notes between the solo and orchestra parts. From our score match, we can compute the solo times for these events, $s_1, s_2, \ldots$, as well as the orchestra times for the events, $o_1, o_2, \ldots$. Note that these sequences only contain the musical positions having *coincident* solo and orchestra notes. In the figure below we have plotted the points (s_k, o_k) and interpolated between them to produce the function $u(t)$.



If we measure the $\{s_k\}$ and $\{o_k\}$ in units of the STFT hop size, H, then the function $u(t)$ in the figure shows how we need to warp the orchestra recording to match the accompaniment. Specifically, we will create our warped version of the orchestra, $\bar{X}$, by

$$\bar{X}(t, n) = X(u(t), n)$$

where X is the straight STFT of the orchestra performance. If $u(t)$ is not an integer for some t, we just round it to the nearest integer and take the corresponding frame of the orchestra STFT. We will get our accompaniment audio by taking the inverse STFT of $\bar{X}$. We will demonstrate several examples of this procedure in class.

Chapter 4

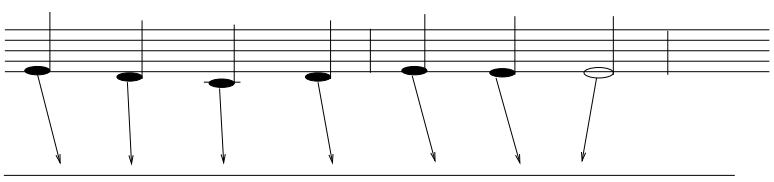
Recognition of Audio and Applications

Score Alignment

The problem of *score alignment* seeks a correspondence between a musical score and an audio performance of that score. For this problem we assume that the musical score is represented in a *symbolic* format, giving, say, the MIDI pitches of the score and the corresponding onset times, in beats or some other unit of musical time. The score may have an approximate tempo, but we assume the performer has considerable latitude in playing the music. For instance, a simple score format representing "Mary had a Little Lamb" format may look like the following, where we have used whole notes as the unit of musical time. This is often a reasonable unit to use since a quarter (or half, eighth, sixteenth, etc.), is one quarter (or half, eighth, sixteenth) of a whole note, regardless of the meter.

onset	MIDI
0/4	64
1/4	62
2/4	60
3/4	62
4/4	64
5/4	64
6/4	64
8/4	62
9/4	62
10/4	62

Our score alignment could be thought of as a third column to this table giving the onset time of the notes in time units such as seconds. This score match is depicted in the figure below.



There are two different kinds of score alignment problems, having different kinds of applications: *Offline alignment* works using the complete audio file. Thus the algorithm can consider both past and future music data in trying to determine the onset time of a particular note. On the other hand, *online alignment*, also known as *score following*, processes the audio data as it comes in, with no potential to consider future music data. This is the appropriate technique when some real-time response to the incoming audio is desired.

What follows is a brief list of possible applications of these problems, though there are many possible additions.

Offline Alignment

Random Audio Access This allows one to hear or see the audio at any position in the music, specified in *musical* units. Think of all the times you have searched for a particular place in a piece of music.

Editing Digital Audio The usual process of making an audio recording may involve some splicing together of various performances or fixing isolated problems individual tracks. A score alignment allows one to easily identify the section of audio corresponding to a particular note or event for further processing.

Coordination of Media Sometimes we may want to align music with some other possible medium, such as video or more audio. The "Oracle" was an example of aligning two different audio performances, in that case a solo and orchestra. For instance, we may want to create an animated dance video that synchronizes to prerecorded music. The score alignment gives the precise times at which the various video events must occur. Another example would be warping video of a conductor to follow a performance.

Quantitative Description of Performance We may want to give a performer or musicologist precise feedback regarding the tuning, timing, or some other aspect of performance. For instance, there has been a good deal of interest in studying timing and the way it relates to musical expression. Another example is a tuning system, designed to show a musician places where tuning inaccuracies occur in the context of a musical score.

Database Collections Sampled sound is now a fundamental element of many (perhaps most) synthetic instruments. Generally, sampled audio is obtained by having a musician play a collection of notes in a highly controlled environment — typically such data is in short supply. Score alignment could be used to mine large quantities of sample data from preexisting recordings.

Online Alignment (Score Following)

Musical Accompaniment Systems These are programs that provide flexible live accompaniment to a musician that synthesizes the missing parts. More on this in what follows.

Content-Based Audio Processing Programs such as *Autotune* have been highly successful in recent years in correcting the tuning in a performance. Live implementations are possible, but made difficult by the uncertainty of knowing the actual note the performer is aiming for. Score following makes the note identifications much simpler.

New Music Applications Similar in spirit to the above, but different in the kinds of applications, one may wish to apply audio effects in a selective score-dependent way. For instance one could harmonize, distort, modify a sequence of notes differently, in real time. This can prove a powerful tool for new music.

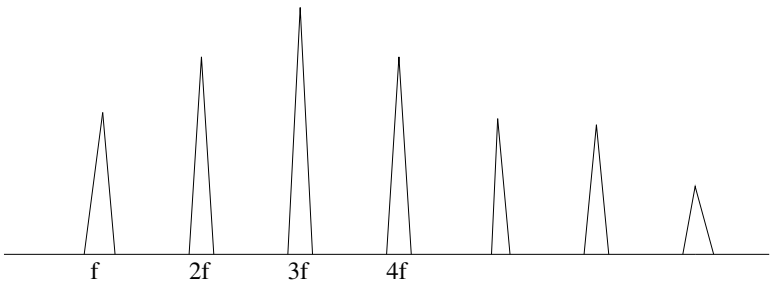
Supertitles or other media such as lights can be coordinated with live performance automatically.

Automatic Page Turners could make the keyboard player's life much easier by turning pages as required.

Relating Audio to Pitch (template_match.r, prob_match.r)

Suppose we have a frame of audio corresponding to a single note from a single instrument. We suppose the pitch is in the set M of the possible MIDI pitches — say the 88 keys on a piano, or the subset of pitches corresponding to the range of the instrument in question. We now look at ways of evaluating the degree to which a pitches "agrees" with the audio data.

The *template matching* approach is a time-honored technique from computer vision, though it appears in other areas as well. The idea is to develop a template, q_m, for each possible pitch, m, giving a representative description of the frequency content for the note. Since the note is pitched, our template would contain peaks in the spectrum at integral multiples of the fundamental frequency, as depicted below:



Suppose that x is the modulus of the FFT of the audio frame and define the "dot" product of x and q_m to be

$$x \cdot q_m = \sum_n x(n) q_m(n)$$

The template matching approach computes the estimated pitch, $\hat{m}$ by

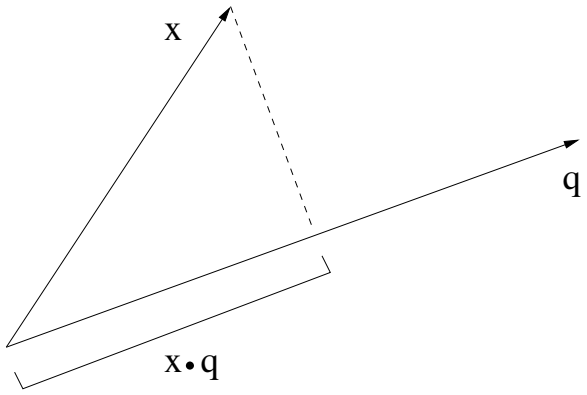
$$\hat{m} = \arg\max_m x \cdot q_m \tag{4.1}$$

That is, we look for the template making the biggest dot product with the audio spectrum, x.

The intuition behind this approach is as follows. Suppose that we let q be a vector having length 1, also known as a "unit vector." This means that $||q|| \stackrel{\text{def}}{=} \sqrt{\sum q^2(n)} = 1$. A familiar identity from linear algebra says that

$$x \cdot q \quad = \quad \text{the length of the projection of } x \text{ onto } q.$$

as in the figure below:



As can be seen from the figure, the dot product gets larger as the vectors "point" in similar directions. Thus if we scale all of our templates to have unit length ($q'_m = \frac{q_m}{||q_m||}$), then choosing the best template match from Eqn. 4.1 is the same as looking for the template q_m that points most nearly in the same direction as x. For this argument to make sense, we must normalize our templates to all have length 1 before using the template match procedure. This approach is demonstrated in the R program **template_match.r**.

A variant on this approach is described by the following simple probabilistic model, though first we introduce a little background in probability.

Suppose we have an experiment that can occur in N ways, $\{1, 2, \ldots, N\}$. Let $p_1, p_2, \ldots, p_N$ be the probabilities of these outcomes having

1. $p_n \geq 0$

2. $\sum_n p_n = 1$

Suppose we observe a sequence of J *independent* trials of the experiment with outcomes $x_1, x_2, \ldots, x_J$, say $x_1 = 3, x_2 = 5, x_3 = 4 \ldots$. The probability of this particular sequence is then

$$P(x_1, \ldots, x_J) = p_{x_1} p_{x_2} \ldots, p_{x_J} = p_1^{\#1's} p_2^{\#2's} \ldots p_N^{\#N's}$$

Now we view our actual spectrum x as a histogram, so that x contains $x(1)$ observations of frequency 1, $x(2)$ observations of frequency 2, etc. We also view our template q_m (scaled to sum to 1) as a probability distribution, so that if pitch m is sounding, x is regarded as a random sample from q_m. Thus the probability of observing x when m is sounding is

$$P(x|q_m) = q_m(1)^{x(1)} q_m(2)^{x(2)} \ldots q_m(N)^{x(N)}$$

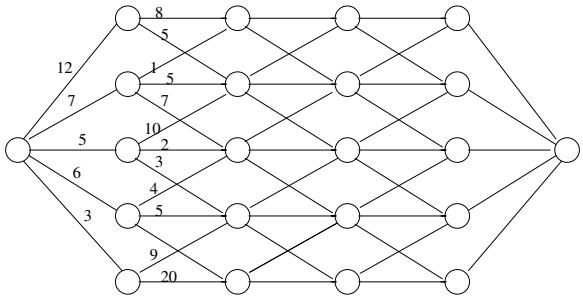
With this probabilistic model, the *maximum likelihood* strategy is to choose the model, q_m giving the greatest likelihood to the data. That is

$$\begin{aligned}
\hat{m} &= \arg\max_m P(x|q_m) \\
&= \arg\max_m \prod_{n=1}^{N} q_m(n)^{x(n)} \\
&= \arg\max_m \sum_{n=1}^{N} x(n) \log q_m(n) \\
&= \arg\max_m x \cdot \log q_m
\end{aligned}$$

Observe that this is very much like our template matching approach except we take the dot product with the log of our template, rather than the template itself. This approach is demonstrated in the R program **prob_match.r**. The probabilistic approach has the advantage that we can choose our templates q_m in a principled way: If we believed the assumptions of the model, a natural estimate for q_m would be just the average of a number of spectra that were generated by pitch m.

Dynamic Programming

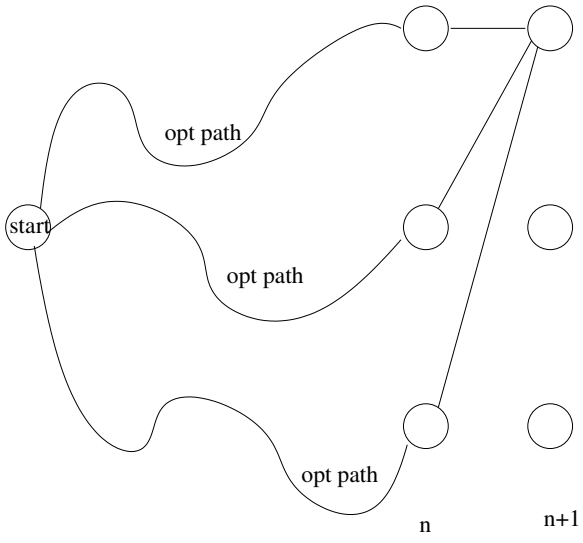
A graph is a construction from computer science and discrete mathematics composed of a collection of *nodes* and *edges*. The nodes are places we might *visit* in the graph, while we can only travel between nodes that are connected by edges. A particular kind of graph is called a *trellis* — while more general definitions are possible, we call a graph a trellis if each of the nodes lives at a level: $0, 1, 2, \ldots, n$ and all edges connect nodes at successive levels. Such a trellis graph is depicted below.



The graph in the figure is known as a *weighted* graph since each of the edges has an associated cost. A common problem one considers with such a graph is to find the minimal cost path from the *start* (leftmost node) to the *end* (rightmost node), where the cost of the path is the sum of the arc costs traversed along the path. This optimal path is computed using a technique known as *dynamic programming*, which relies on a single simple, yet powerful, observation as follows:

For each node, n, in the trellis, the best scoring path to the node is a best scoring path to a node, m, at the previous level, plus the arc (m, n).

This is clearly true since if we were to consider a suboptimal path from the start to m plus the arc (m, n), clearly this achieves a worse score than the optimal path to m plus (m, n). This idea is illustrated in the following figure where the optimal path to a node a level n is seen to be an extension of an optimal path to a node an level n:



The observation leads directly to an algorithm for computing the optimal path. Let $c(i, j)$ be the cost realized in going from node i to node j. Let $m(i)$ be the cost of the optimal path from the start to node i and set $m(\text{start}) = 0$. Then, reasoning from our "key observation," we see that the score of the optimal path to node j must be the *best* of the optimal paths to the predecessors, i, of j with (i, j) concatenated onto these optimal paths. Put algorithmically, we can compute the score of the optimal path from start to end by letting

$$m(j) = \min_{i \in \text{pred}(j)} m(i) + c(i, j)$$

where pred(j) is the *predecessors* of j — the collection of notes that are connected to j "from the left." We will compute the costs $m(j)$ from left to right in our graph. Then, when these optimal costs are all computed, m(end) will be the optimal cost from start to end.

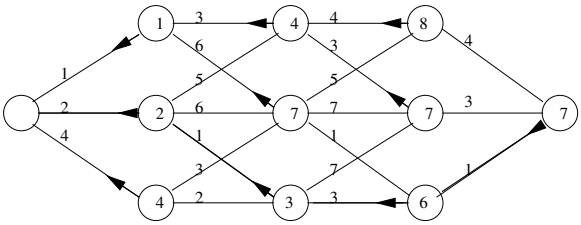
Of course, we really wanted to find the optimal *path*, not just its cost. A simple addition to our algorithm solves this problem. Let

$$a(j) = \arg \min_{i \in \text{pred}(j)} m(i) + c(i,j)$$

so that $a(j)$ is the optimal predecessor of node j. When we have computed $a(j)$ and $m(j)$ for *all* trellis nodes, we can then "trace back" the optimal path by following the optimal predecessors back from the end node. That is, the optimal path (in reverse order) is given by:

$$\text{end}, a(\text{end}), a(a(\text{end})), a(a(a(\text{end}))), \ldots, \text{start}$$

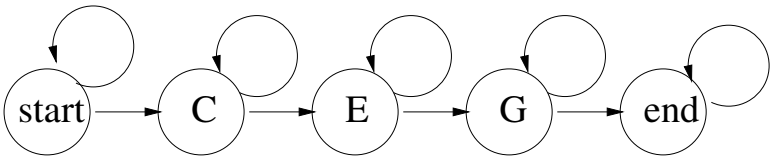
In the following simple example we have filled in the circles of each node j with the optimal cost to j, $m(j)$, and have indicated the optimal predecessors of each node with an arrow that points backward along the appropriate arc.



One last comment. We have presented DP as a tool for finding the minimal cost path through a trellis graph. However, if each arc has some sort of "score" attached to to, and we wish to find the highest scoring path, we simply replace minima by maxima in the expressions for $a(j)$ and $m(j)$.

Dynamic Programming Approach to Score Following

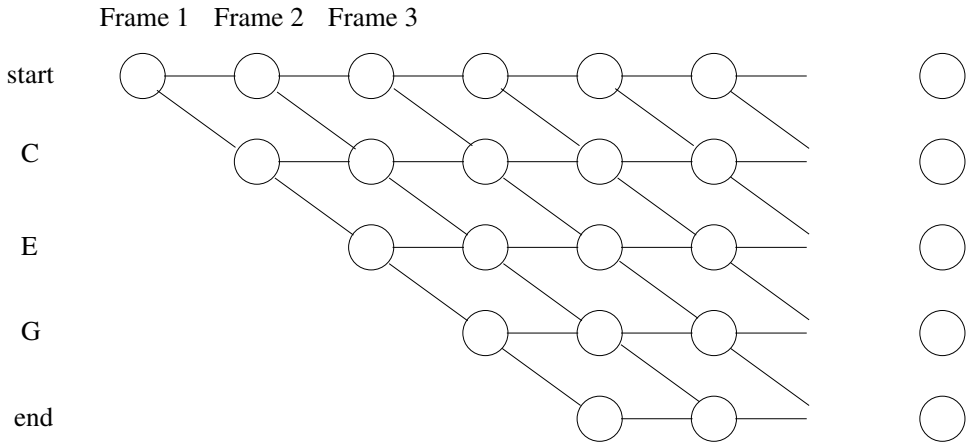
Suppose we have a score made out of three notes: C,E,G. We will ignore the rhythm associated with the notes and consider the 5-state graph as follows:



We think of this graph as describing the possible sequences of frame labellings we are willing to consider. Note that every sequence that this graph can generate looks like

$$\text{start}, \ldots, \text{start}, C, \ldots, C, E, \ldots, E, G, \ldots, G, \text{end}, \ldots, \text{end}$$

This same graph can also be represented as a trellis graph where the tth layer of the trellis corresponds to the possible labellings of the tth audio frame. This graph is shown in the figure below.



If we label the rows of the trellis as start, C, E, G, end, as indicated in the figure, then the paths through our trellis graph create the same sequences of frame labellings is in our original 5-state graph. This is the crux of the matter and why the approach we are presenting is so far superior to the frame-by-frame labeling approach we first introduced: The trellis graph constrains the possible sequences of frame labels to be *meaningful* sequences for score alignment.

We now want to cast our score matching problem as a search for the optimal path through our trellis graph, and to do this we must assign scores to the arcs of the trellis. To do this, the score we will give an arc that terminates in a node in row r at column t will be

$$\log P(X(t,\cdot)|\text{model } r)$$

where model 2,3,4 are the C,E,G models while models 1,5 is a rest model. (In this latter case of a rest we may model the spectrum as a sample from a uniform distribution in which all frequencies are equally likely.)

Now we can find the most likely path through this graph using dynamic programming, as discussed before. If $l_1, l_2, \ldots, l_T$ is a labeling of the entire collection of T frames, then we are looking for the best scoring labeling of sequences from the 5-state model, where the score of a labeling is

$$\sum_{t=1}^{T} \log P(X(t,\cdot)|l_t)$$

Implementation

This idea can be implemented with a pair of $5 \times T$ arrays, one we will call the Score array and the other the Pred array. For these arrays

Score[j,t] is the score of the optimal path through the trellis ending in state j at frame t.

Pred[j,t] is index (in $1 \ldots, 5$) of the optimal predecessor of state (j, t).

In terms of our previous discussion, Score[j,t] is the number we wrote inside the circle representing a trellis node, while Pred[j,t] is the arrow that pointed backward to the previous optimal state.

Since this algorithm is especially important to us, we present here the R code needed to compute both the Score and Pred arrays. We begin by initializing the 1st column of the Score array so that

$$\text{Score}[j,1] = \begin{cases} 0 & \text{if j} = 1 \\ -\infty & \text{otherwise} \end{cases}$$

meaning that we force the path to begin in the *start* state. Then the following algorithm would fill in the remaining columns of the array in succession:

```
for (t in 2:T) {                              # loop over all frames 2 ... T
  for (j in 1:notes) {                        # loop over all notes
    if (j == 1) parents = 1;
    else parents = c(j,j-1);                  # the possible prececessors of the state
    Score[j,t] = -Inf;                        # initialize the score to -infinity
    lp = log(frame t | state j)               # not really R code, but you get the idea
    for (jj in parents) {                     # loop over all predecessors
      if (Score[jj,t-1] + lp > Score[j,t]) {  # found a better predecessor
        Score[j,t] = Score[jj,t-1] + lp;
        Pred[j,t] = jj;
      }
    }
  }
}
```

Having executed this algorithm, we have computed the score of the best possible path to every node in our trellis — in particular, the node (5,T), which we must occupy at the final frame. Recall hat we trace back the optimal path by "following the arrows backward" in our annotated trellis graph. That is, if we had $T = 10$ frames and our Pred array read as:

$$\begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ - & 2 & 2 & 2 & 2 & 1 & 2 & 2 & 2 & 2 \\ - & - & 3 & 2 & 3 & 3 & 3 & 2 & 2 & 2 \\ - & - & - & 4 & 4 & 3 & 3 & 4 & 3 & 4 \\ - & - & - & - & 5 & 5 & 4 & 5 & 4 & 4 \end{array}$$

we would trace back the optimal path as follows. We note we must end in state 5 at frame T=10. Since Pred[5,10] = 4, we know that 4 is the optimal predecessor, so the last two states of the optimal sequence will be 4,5. Since Pred[4,9] = 3 is the next optimal predecessor, we know the last three states are 3,4,5. Continuing in this manner we unravel the sequence of optimal states as

$$1, 1, 1, 1, 1, 2, 2, 3, 4, 5$$

This says our score match is given by 5 frames of rest, followed by 2 frames of C, followed by one of 3, one of 4, and one of rest.

From an algorithmic perspective, we would code the optimal traceback as

```
hat = rep(0,10);           # will hold the optimal sequence of labels
hat[10] = 5;               # must have this label for last state
hat[9] = Pred[hat[10],10]
hat[8] = Pred[hat[9],9]
hat[7] = Pred[hat[8],8]
etc.
```

or more generally as

```
hat = rep(0,T);           # will hold the optimal sequence of labels
hat[T] = 5;               # must have this label for last state
for (t in T:2)
  hat[t-1] = Pred[hat[t],t]
```

This technique is demonstrated in the R program **score_follow.r**.

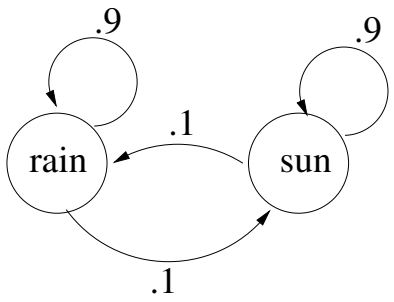
Markov Models

A *Markov Chain* is a sequence of random variables $x_1, x_2, \ldots$ taking values in some finite state space $\Omega = \{\omega_1, \ldots, \omega_L\}$ such that

$$p(x_t|x_1, \ldots, x_{t-1}) = p(x_t|x_{t-1})$$

That is, each state depends only on the previous state.

Often a Markov chain is depicted with a *state graph* showing the states as nodes in a graph with directed (with arrow) edges showing transition probabilities. The sum of the probabilities of all edges leaving a state must be 1. If we denote sun by S and rain by R, a sequence generated by this Markov chain would look like *SSSSSRRRRRRRRRRSSSSSSSRRRR*



To make our definition more clear, consider the case of four random variables x_1, x_2, x_3, x_4. No matter what kind of dependence these variables have, it is always true that

$$p(x_1, x_2, x_3, x_4) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)p(x_4|x_1, x_2, x_3)$$

In the case of a Markov Chain, these factors simply to:

$$p(x_1, x_2, x_3, x_4) = p(x_1)p(x_2|x_1)p(x_3|x_2)p(x_4|x_3)$$

More generally, if $x_1, x_2, \ldots, x_N$ is a Markov chain, then

$$\begin{aligned} p(x_1, \ldots, x_T) &= p(x_1)p(x_2|x_1)p(x_3|x_2)\ldots p(x_T|x_{T-1}) \\ &= p(x_1)\prod_{t=2}^{T} p(x_t|x_{t-1}) \end{aligned}$$

As a result of this, we see that in order to define a Markov chain we need

1. The *initial* distribution $p(x_1)$
2. The *transition probability matrix* Q where $Q_{ij} = p(x_{n+1} = \omega_j | x_n = \omega_i)$

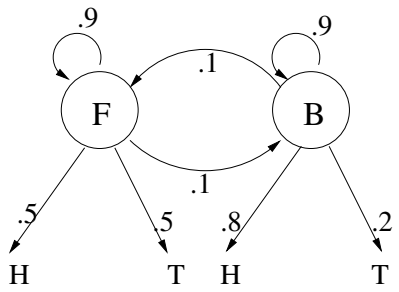
For instance, in the "Rain and Sun" Markov chain introduced above, we would have two states $\Omega = \{\omega_1, \omega_2\} = \{\text{Rain}, \text{Sun}\}$ where

$$Q = \begin{pmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{pmatrix} = \begin{pmatrix} p(\omega_1|\omega_1) & p(\omega_2|\omega_1) \\ p(\omega_1|\omega_2) & p(\omega_2|\omega_2) \end{pmatrix} = \begin{pmatrix} .9 & .1 \\ .1 & .9 \end{pmatrix}$$

Hidden Markov Models

Suppose we have a Markov chain, $x = x_1, x_2, x_3, \ldots$, but *do not* observe x directly. Instead, each time we visit a state, the state "outputs" a random observation from some distribution associated with the state.

As an example, consider the following figure that describes an observed sequence of coin flips. In this example there are two coins, a "fair" coin, F, and an "biased coin, B. The fair coin gives equal probability to H (heads) and T (tails), but the biased coin gives probability .8 to H (and .2 to T). The coin flipper flips the current coin and then makes a random choice of which coin to use next. In doing this, she stays with the current coin with probability .9 and switches to the other coin with probability .1.



An observed sequence from the coin may looking something like the following.

$$\underbrace{THHTHHTTHHTT}_{\text{fair?}}\underbrace{HHHTHHTHHHHHHH}_{\text{biased?}}\underbrace{THTHHTH\ldots}_{\text{fair?}}$$

In hidden Markov model problems (HMMs), the inference problem is usually to give meaning to the observations by uncovering the sequence of hidden states. This meaning (here which coin is operating) is annotated in the equation above for the particular sequence of flips.

More formally, we assume that each observation, y_n, depends only on the current state, x_n. We will write x and y for the entire sequences of states and observables: $x = (x_1, x_2, \ldots, x_T)$, $y = (y_1, y_2, \ldots, y_T)$. Thus

$$\begin{aligned} p(y|x) &= p(y_1, \ldots, y_T|x_1, \ldots, x_T) \\ &= p(y_1|x_1, \ldots, x_T)p(y_2|y_1, x_1, \ldots, x_T)\ldots p(y_T|y_1, \ldots, y_{T-1}, x_1, \ldots, x_T) \\ &= p(y_1|x_1)p(y_2|x_2)\ldots p(y_T|x_T) \end{aligned}$$

Putting this together with the factorization of the Markov chain, x, gives

$$\begin{aligned} p(x,y) &= p(x)p(y|x) \\ &= p(x_1)p(x_2|x_1)p(x_3|x_2)\ldots p(x_T|x_{T-1}) \\ &\quad \times p(y_1|x_1)p(y_2|x_2)p(y_3|x_3)\ldots p(y_T|x_T) \\ &= \underbrace{p(x_1)p(y_1|x_1)}_{f(x_1)}\underbrace{p(x_2|x_1)p(y_2|x_2)}_{f(x_1,x_2)}\underbrace{p(x_3|x_2)p(y_3|x_3)}_{f(x_2,x_3)}\ldots\underbrace{p(x_T|x_{T-1})p(y_T|x_T)}_{f(x_{T-1},x_T)} \end{aligned}$$

Usually in an HMM the y's are *observed* and hence are known to us. On the other hand, the x's are *unobserved*, and hence unknown. Thus, we can think of this factorization as a product of functions where the first, $f(x_1)$, just depends on the first state variable, the 2nd, $f(x_1, x_2)$, depends on the first two state variables, the third, $f(x_2, x_3)$, depends on the next two state variables, etc. This view of the factorization will be important to us in what follows.

HMMs for Recognition (coin_hmm.r)

Suppose we have an HMM, (x, y), with $p(x, y)$ as before. We *observe* $y = (y_1, y_2, \ldots, y_T)$ and want to find the *most likely* state sequence given our observations. That is, we seek $\hat{x} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_T)$ where

$$\hat{x} = \arg\max_x p(x|y)$$

Note that since $p(x|y) = \frac{p(x,y)}{p(y)}$ and y is *fixed*, we have

$$\begin{aligned} \hat{x} &= \arg\max_x p(x|y) \\ &= \arg\max_x p(x,y) \\ &= \arg\max_{x_1,\ldots,x_T} \underbrace{[p(x_1)p(y_1|x_1)]}_{f(x_1)}\underbrace{[p(x_2|x_1)p(y_2|x_2)]}_{f(x_1,x_2)}\ldots\underbrace{[p(x_T|x_{T-1})p(y_T|x_T)]}_{f(x_{T-1},x_T)} \end{aligned}$$

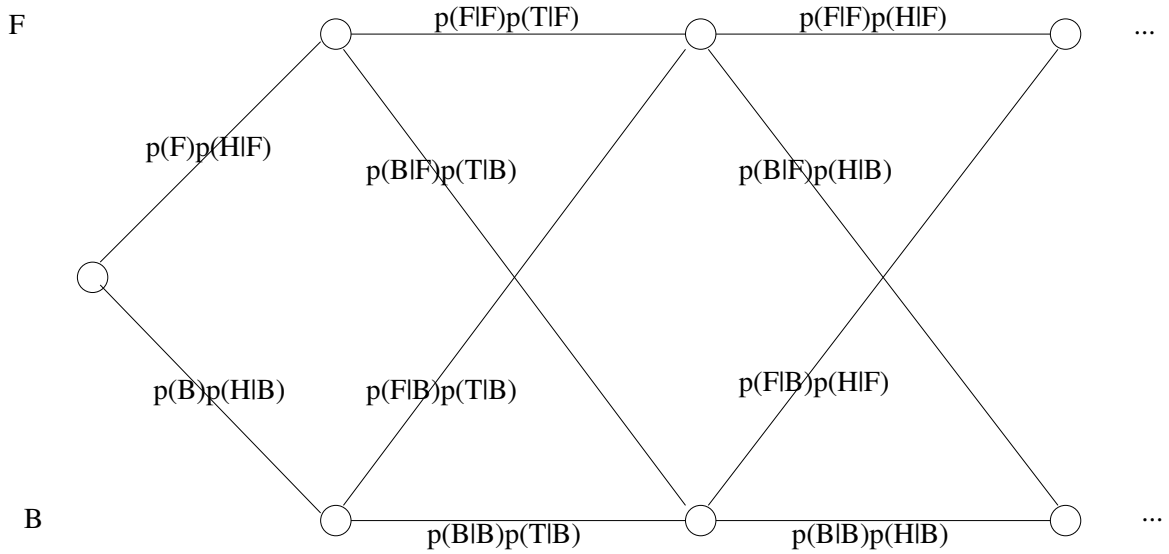


Figure 4.1: The trellis for the coin-flipping HMM with fair and biased coins.

The important thing to notice about this factorization is that we express the quantity we want to optimize, $p(x, y)$, as a product of factors depending on *consecutive* variables from the chain (x_t, x_{t+1}). This is ideal for maximizing using dynamic programming, as follows.

Suppose we observe an T-long sequence $y_1, \ldots, y_T$ and our state variables x_t can take values in a state space $\Omega = \{\omega_1, \ldots, \omega_L\}$ having L different states. We create a trellis graph having T "levels" with L possible states at each level — thus we can think of the trellis as an $L \times T$ array of states. We define the initial score for the state l in layer $t = 1$ as $p(x_1 = \omega_l)p(y_1|x_1 = \omega_l)$. Then we define the transition score for moving from state l at stage $t - 1$ to state l' at stage t as $p(x_t = \omega_{l'}|x_{t-1} = \omega_l)p(y_t|x_t = \omega_{l'})$

Suppose now that we take the score of an entire path through the lattice as the *product* of the scores traversed along the arcs. (In the past we took the sum of arc scores, but this is just a minor difference). Then for any path through the lattice, (that is, any T-long sequence of states), $x_1, x_2, \ldots, x_T$, the score of this path is

$$p(x, y) = [p(x_1)p(y_1|x_1)][p(x_2|x_1)p(y_2|x_2)] \ldots [p(x_T|x_{T-1})p(y_T|x_T)]$$

where each factor inside the brackets represents a transition score at the associated level of the trellis graph. Thus we have constructed a weighted graph where, for each path $x = x_1, \ldots, x_T$, the score of this path is $p(x, y)$. Since we can use dynamic programming to find the best scoring path through the trellis, we will have also found the state sequence that maximizes $p(x, y)$.

This is best illustrated in terms of an example, so let's return to the simple HMM describing the fair and biased coins. Suppose that we observed the sequence $y = HTH \ldots$ and consider Figure 4.1.

Note that the first flip is H, so the score for beginning in the fair state, F (moving from the start state to F), will be $p(F)p(H|F)$ while the score for beginning in the biased state (moving from the start state to B) will be $p(B)p(H|B)$. This reflects our prior probabilities for beginning in the two states, $p(F)$ and $p(B)$, as well as the likelihood for observing our first flip, which was H, in the two states, $p(H|F)$ and $p(H|B)$.

Then in the second level of the trellis there are four possible transitions to consider, $F \to F$, $F \to B$, $B \to F$,$B \to F$, each with a prior probability, $p(F|F)$, $p(B|F)$, $p(F|B)$, $p(B|B)$. Since our second observed flip was T, we must also factor in the observation probability associated with the transitions. Thus, if we move to state F (from either preceding state) the observation probability will be $p(T|F)$ while if we move to state B the probability will be $p(T|B)$. These are exactly the transition scores that are indicated in the Figure 4.1.

Finally, there is one more trellis level sketched for the 3rd observation, H. This follows exactly the same pattern as we observed before except that our observation probabilities will be for H rather than T, as in the previous trellis level. Note that the product of all of the score traversed along any path such as $x = BBBBBFFFFFBBBB \ldots$ will be $p(x, y)$. Thus, we can find the most likely sequence given our observations using dynamic programming. This idea is implemented for the coin flipping example in **coin_hmm.r**.

The dynamic programming will nearly identical to the way we originally introduced the idea. The scores for the initial state, F, B will be $s_1(F) = p(F)p(H|F)$ and $s_1(B) = p(B)p(H|B)$ as indicated in the figure. We have also described the way we can construct transition scores, $s_n(F,F), s_n(F,B), s_n(B,F), s_n(B,B)$ for moving through the various state pairs on the nth level. Since there is only one possible path leading to the states at the first level, we will define the optimal scores for these as $m_1(F) = s_1(F)$ and $m_1(B) = s_1(B)$. But for subsequent levels we got the scores of the optimal paths as

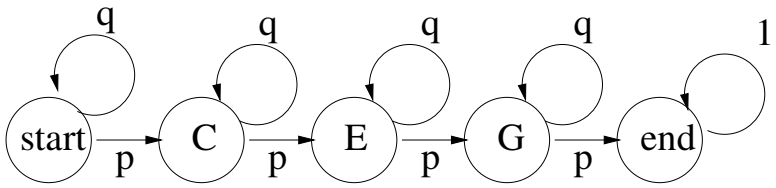
$$\begin{aligned} m_n(F) &= \max(m_{t-1}(F)s_n(F,F), m_{t-1}(B)s_t(B,F)) \\ m_n(B) &= \max(m_{t-1}(F)s_n(F,B), m_{t-1}(B)s_t(B,B)) \end{aligned}$$

Thus, $m_T(F)$ and $m_T(B)$ will be the scores of the optimal paths ending in F and B. If one of these is larger than the other, it will be the optimal score through the graph which is the same as $\max_x p(x,y)$. Of course, we are interested in finding the best sequence, not the probability it creates, so we need to remember how we constructed the optimal score. To this end we simply need to remember which which predecessor state creates the optimal score at level t. This is done exactly as before by remembering the optimal predecessor to each trellis state. This approach is demonstrated in the R program **coin_hmm.r**.

Score Alignment with HMMs

The HMM framework allows us to extend and improve the basic approach to score following that relies on dynamic programming, already presented. To do this we must be explicit about the nature of our model states, the transition probabilities between the states, and the output probabilities of our observable data (the frame spectra) given the states.

The simplest approach would be to use one state for each note of the score, as well as a start and end state as we did before. Then if we let p be the probability of moving to the next note and $q = 1 - p$ the probability of remaining in the current state, then the state graph for our Markov chain looks as follows:



This model gives the probability of any sequence of labels from $\{\text{start}, C, E, G, \text{end}\}$ as the product of transition scores.

With this model we can use the simple output model we used before:

$$P(Y_t = y_t | X_t = r) = \prod_{n=1}^{N} q_r(n)^{y_t(n)}$$

where q_r is the template for the rth score note, normalized to 1 and viewed as a probability distribution and y_t is the tth audio spectrum, viewed as a histogram of "counts."

In truth, this model simply casts our original score follower as an HMM — since the HMM version optimizes the identical objective function, it will perform identically. However, there are a number of variations of this idea that only make sense within the HMM framework. We introduce several of these here.

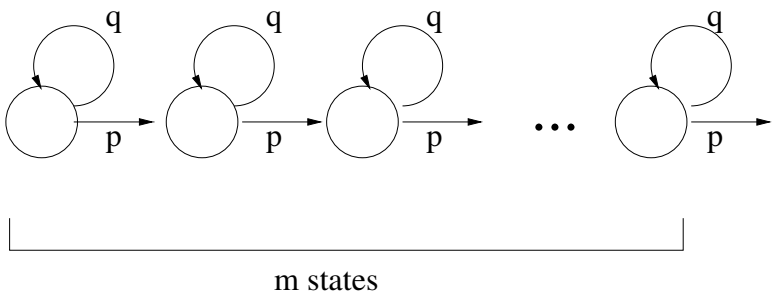
The first of these concerns the model on note lengths. Implicit in the simple one-state note models we used for our Markov chain is an Exponential distribution on the length of each note. That is, consider the number of frames, L, we spend in the note model marked as C above. Since each frame we either return to the C state with probability

q or move on with probability p, we have

$$\begin{aligned} P(L=1) &= p \\ P(L=2) &= qp \\ P(L=3) &= q^2p \\ &\vdots \quad \vdots \quad \vdots \\ P(L=l) &= q^{l-1}p \end{aligned}$$

This distribution is often called "Exponential" or "Geometric." While it may be convenient from a mathematical perspective, it is certainly not a reasonable model for an actual note length. Imagine a musician who decides at a rate of 30 times a second if she will continue to the next note by flipping a biased coin. While this model *does* capture the uncertain nature of the situation, the particular kind of uncertainty modeled doesn't seem musically reasonable.

A more plausible model chains together a collection of m states as in the figure below:



In calculating the distribution on the number of frames spent in this model, let us consider the probability $P(L=l)$ — the probability we spend l frames in this model. For $L=l$ to occur, the lth transition must move us out of the model, which happens with probability p. For the remaining $l-1$ transitions, $m-1$ must move between states (p) while the remaining $l-m$ must state within a state (q). Thus the probability of any particular path that spends l frames in the model will be $p \times p^{m-1} \times q^{l-m} = p^m q^{l-m}$. There are $\binom{l-1}{m-1}$ such paths since, of the last $l-1$ transitions, we can choose $m-1$ of them to be p transitions with the remaining ones q transitions. Thus we have

$$p(L=l) = \binom{l-1}{m-1} p^m q^{l-m}$$

$l = m, m+1, \ldots$ This distribution is known as the "Negative Binomial" distribution. A simple calculation shows that the mean of this distribution is m/p and the variance is mq/p^2. We can choose the number of states in the model, m, and the parameter p, so that the resulting mean and variance model our belief about this note length distribution. For instance, the mean value could be the length of the note in frames, using the nominal tempo. Better yet, the mean and variance can be estimated from past examples. Estimating the parameters m and p in this way, by matching the model moments (mean and variance) to the empirical of observed moments, is known as the *method of moments*. This is the strategy we employ in practice.

Another improvement is to include features other than our spectrogram. For instance, note attacks are often signaled by a sudden burst of energy in the audio data. For repeated pitches, there is virtually no way to identify note boundaries without identifying some kind of onset event. To do this, we could create a feature that measures this "burstiness" and add on an extra state or two to the beginning of our note model to model this attack. Presumably when we visit the attack state, we expect to see more burstiness in the data, while we anticipate less in the remaining states. This would be captured by using a different output distribution on the burstiness feature for the start of the note than is used for the other states.

An additional improvement performs the score match not with the optimal path, as we have used until now, but using the so called forward-backward labeling. While we will omit the details, for a HMM one can compute

$$p(x_t = s|y_1, \ldots y_T)$$

That is, the distribution on the tth state variable given *all* of the observable data. This probability distribution represents our knowledge about where we are in the Markov model at the tth frame of audio data, having observed all of the audio data. We can then recognize the onset of the kth note as the frame that is most likely to be in the first state of the note:

$$\hat{o}_k = \arg\max_t p(x_t = \text{start}_k|y_1, \ldots, y_T)$$

Such a scheme performs better for score following.

There are numerous other improvements that can be made to the score alignment HMM, however, in an attempt to avoid getting bogged down with details, we won't describe these here. Instead, we will present in class a number of actual examples of score alignment on highly challenging audio data. This gives a sense of what is possible in this domain.

Online Score Following

As we have seen, not all inference with Hidden Markov models is necessarily based on the most likely state sequence $\hat{x} = \arg\max_x p(x|y)$. For example, in the *filtering* problem we compute the distribution on the current state variable given the past observations:

$$p(x_t|y_1, y_2, \ldots, y_t)$$

This so-called *filtered* probability expresses what we know about the current state of the Markov chain given what we have seen so far. In some cases we might be reasonably certain about what state we are in — in this case the filtered distribution will concentrate most of its mass on a single state. In other cases we may be highly uncertain of the whereabouts of the chain — in this case the filtered probability may be quite spread out. The ability to make this distinction is an advantage of probabilistic modeling, since we may be more inclined to take action when we are certain, yet prefer to wait when we are less certain (remember the boy who cried "wolf").

If we let

$$\alpha_t(i) = p(x_t = i|y_1, \ldots, y_t)$$

we can compute α_t through the following recursion:

$$\alpha_1(i) = \frac{p(x_1 = i)p(y_1|x_1 = i)}{\sum_{i'} p(x_1 = i')p(y_1|x_1 = i')} \tag{4.2}$$

$$\alpha_{t+1}(i) = \frac{\sum_j \alpha_t(j)p(x_{t+1} = i|x_t = j)p(y_{t+1}|x_{t+1} = i)}{\sum_{i'}\sum_j \alpha_t(j)p(x_{t+1} = i'|x_t = j)p(y_{t+1}|x_{t+1} = i')} \tag{4.3}$$

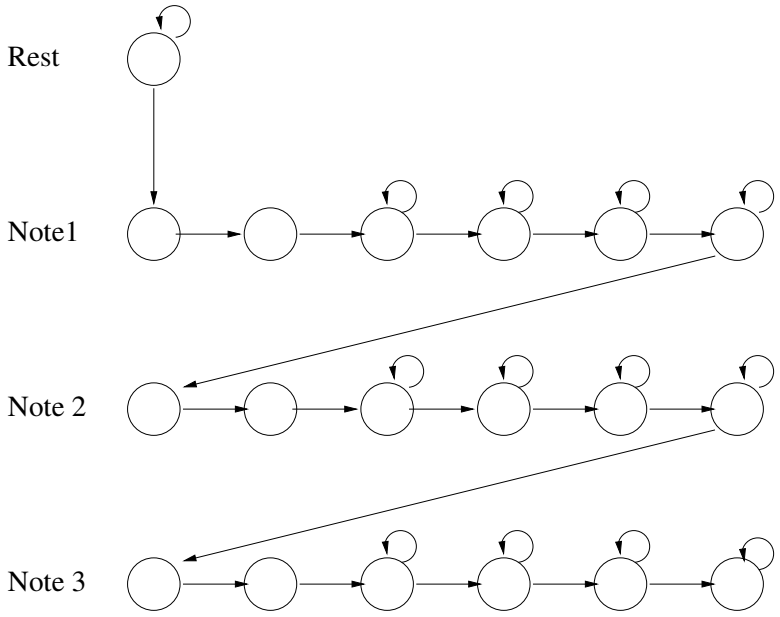
In this expression for $\alpha_{t+1}(i)$ the denominator is just the normalizing constant needed to make the expression a probability distribution — though both i' and j appear in this expression, these variables are just summation indices and not actual variables like i in the numerator.

To see the correctness of the formula, note that by simple rules of probability

$$\begin{aligned}
\alpha_{t+1}(i) &= p(x_{t+1} = i|y_1, \ldots, y_t, y_{t+1}) \\
&= p(x_{t+1} = i, y_{t+1}|y_1 \ldots, y_t)\frac{p(y_1, \ldots, y_t)}{p(y_1, \ldots, y_{t+1})} \\
&\propto p(x_{t+1} = i, y_{t+1}|y_1, \ldots, y_t) \qquad \text{(viewed as a function of } i\text{)} \\
&= \sum_j p(x_{t+1} = i, x_t = j, y_{t+1}|y_1, \ldots, y_t) \\
&= \sum_j p(x_t = j|y_1, \ldots, y_t)p(x_{t+1} = i|x_t = j, y_1, \ldots, y_t)p(y_{t+1}|x_t = j, x_{t+1} = i, y_1, \ldots, y_t) \\
&= \sum_j \alpha_t(j)p(x_{t+1} = i|x_t = j)p(y_{t+1}|x_{t+1} = i)
\end{aligned}$$

Since $\alpha_{t+1}(i)$ is a probability distribution it must sum to 1. Thus if we divide the right-hand-side by the appropriate constant to make it sum to 1 we have equality rather than just proportionality. Eqn. 4.3 simply adds appropriate denominator to make this happen.

To imagine this formula in action in the score following context, consider an HMM with Markov chain as in the following figure:



Each iteration of the *forward algorithm* (the calculation of the α_t probabilities) advances one step by computing

$$\alpha_{t+1}(i) = \frac{\sum_j \alpha_t(j) Q_{ji} R_{iy_{t+1}}}{\text{normalizing constant}}$$

where, in our simplified notation, Q_{ji} is the transition probability of going from state j to state i and $R_{iy_{t+1}}$ is the probability of observing the $t+1$th frame of data, y_{t+1} given we are in state i.

We will initialize our model with $p(x_1 = \text{init rest}) = 1$ so that for the first frame of audio $\alpha_1(i)$ concentrates all of its probability mass on the initial rest state. Imagine now that we start by receiving a sequence of frames that are consistent with the rest state. This means that R_{iy_t} is high when i is the rest state but lower otherwise. In this case, even though the α_t probabilities generate hypotheses that move forward through the state graph, after multiplying by the data probability, only the hypothesis of the rest state will have any significant amount of probability mass. Thus $\alpha_n(i)$ will continue concentrating its probability mass on the rest state as long as our audio data are consistent with the rest assumption.

Eventually we will see audio data that is consistent with the first note of the piece. When this first happens the state at the start of the 1st note will have significant probability after the forward iteration is computed. As we continue to get audio data consistent with the first note, probability mass begins to propagate through the first note. Since the first two transitions of the note are *deterministic* (have probability 1) we know that two frames later all of the probability mass will have moved forward to the first state with the self-loop of note 1. As we continue to get audio data consistent with the 1st note, this mass propagates forward through the chain of self-loops. Since all of these state have the same observation probability, this propagation only depends on the transition probabilities. If p is the probability of moving forward, p percent of the probability mass of each state moves to the next state in each iteration. One can think of this as a "bulge" that moves through the sequence of states until, after the appropriate number of frames, it begins "knocking on the door" of the next note. Thus the model is poised to recognize the

next note after the first note has sounded for the appropriate amount of time. During this period if we get modest evidence of the 2nd note, probability mass will easily slide into the 2nd note model. However, if the audio data confirms that we are still in the first note, the probability mass that sneaks into the 2nd note will be decreased to nothing by the output probabilities.

Thus probability mass continues to move through our network of state, most easily transitioning from one note to the next after an appropriate amount of time spent in the current note, but always bound to be consistent with the audio data.

Our score follower provides a running commentary on the audio data, pinpointing the onsets of notes and communicating these estimates during the performance. This works as follows. Suppose we have observed notes $1, 2, \ldots, k-1$ and are currently waiting to see the kth note. We continue computing the forward probabilities until

$$p(x_t \geq \text{start}_k | y_1, \ldots, y_t)$$

is sufficiently large where start_k is the first state of the kth note. That is, we wait until we are reasonably certain that we have passed the onset of the kth note. Note that this computation is easily done using the α probabilities — we simply add up all α probabilities for states that are beyond the start of the kth note. Eventually this probability will be large enough for us to be confident that the kth note has sounded — say that t^* is the time (frame) at which this level of certainty occurs.

At this point we look backwards in time (look at $t \leq t^*$) and for each such frame compute

$$p(X_t = \text{start}_k | y_1, \ldots, y_{t^*})$$

Note that we are using all of the data up to t^* to do this. While we haven't discussed the mechanics of this computation, it is easy to do with HMMs. Intuitively, this is exactly what we want to do. Having determined that the onset of the kth note is in the past, let's look backward to see where we think it began. While many estimation techniques are possible, we choose the frame t where the note is most likely to begin. That is we estimate the onset of the kth note by

$$\hat{o}_k = \arg\max_{t \leq t^*} P(X_t = \text{start}_k | y_1, \ldots, y_{t^*})$$

In class we will show a real-time demonstration of the way in which this calculation works. It is important to note that there will always be latency built into this estimation problem, since we can't detect a note until we already have some evidence for its existence.

Musical Accompaniment Systems

A musical accompaniment system is a program that provides an accompaniment to a live player in a piece of music for soloist and accompanying ensemble. Our particular flavor of the problem deals with non-improvisatory "classical" music such as a concerto for violin and orchestra.

The problem can be motivated by the familiar "Music Minus One" play-along records that provide (mostly) orchestral accompaniments to familiar solo repertoire. While these records are a nice introduction to the experience of the orchestral soloist, they do not capture the experience very faithfully. Common musical thinking dictates that the orchestra should follow the soloist, which, of course, cannot happen with a recorded accompaniment. One can think of the musical accompaniment system as trying to turn this around so the orchestra follows the soloist.

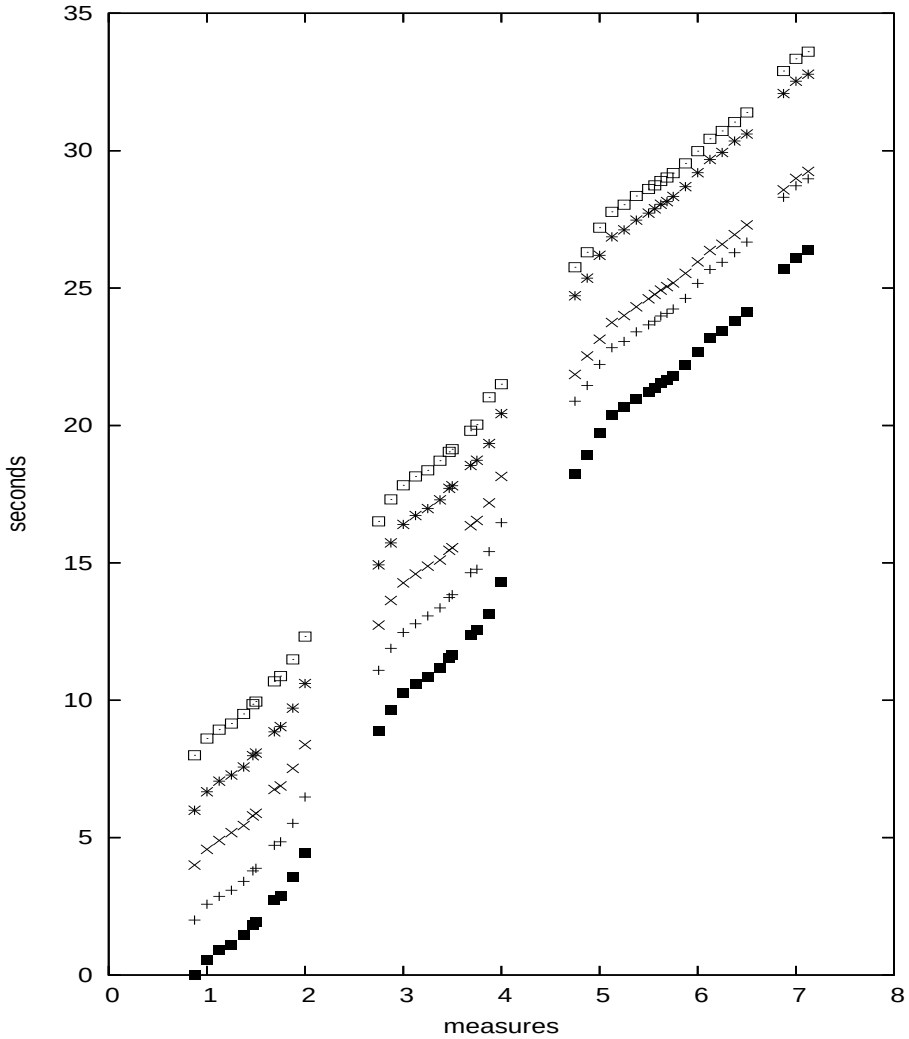
There are three principal ingredients to such as system:

1. Real time score following
2. Phase vocoded audio output (or midi output)
3. Accompaniment control (the brain of the system).

We have already talked about real time or online score following. We have also talked about phase vocoding, which can be used to generate an audio accompaniment by warping the playback rate to match the soloist. Of course, there are other ways that the accompanying sound can be generated. The final component is a musical brain that uses the note onset times provided by the score follower to drive the performance of the accompaniment. While this subject doesn't obviously fit into a class on audio, it doesn't seem to belong in any other class I can think of either. So I will discuss it here.

Consider first the simple-minded approach to accompaniment that works as follows. The orchestra sees that it needs to synchronize a particular note with an note from the soloist. It listens intently for the note and when it is finally detected it plays the orchestra note. This method seems destined for failure. It is not possible to detect a note until it has sounded for a bit of time, thus any correct note detections must be made some time after the note has begun. Thus, if the orchestra is merely going to "respond" to the detection event, it will necessarily be late. In practice, the amount of detection *latency* — the lag between the actual start of the note and the time at which it is detected — is musically significant and generally proves nearly fatal in any music with a feeling of steady pulse. In light of this observation we will base our system on *prediction*, modeling something like a human approach. That is, our system will continually predict the future evolution of the orchestra part based on what it has currently seen. This prediction will be continually modified as more information is made available.

First let's consider some actual timing data from real music. The following figure shows the times, in seconds, of the note onsets of the performance, plotted against the musical times, in measures, of the notes.



In the figure each of the performances are "stacked" vertically for easy comparison. If such a performance were played completely in time, with each actual note length proportional to the note value given in the score, then the points would lie on a line. Clearly this is not the case. This particular example shows a common syndrome in musical performance: the tempo is gradually increased over the first several notes to "ease into" the basic tempo, while the tempo slows down at the end of the phrase. This non-metronomic aspect makes it particularly difficult to follow such a musical interpretation. However, one can also see that there is a good deal of commonality between the performance, so that we may hope to learn from past performances in predicting the timing evolution of the current performance.

Here is a possible model for the musical timing in a piece of music. We let t_k denote the time, in seconds, that the kth note begins and s_k be the tempo, in seconds per beat, for the kth note.

$$\begin{aligned} s_{k+1} &= s_k + \sigma_k \\ t_{k+1} &= t_k + l_k s_k + \tau_k \end{aligned}$$

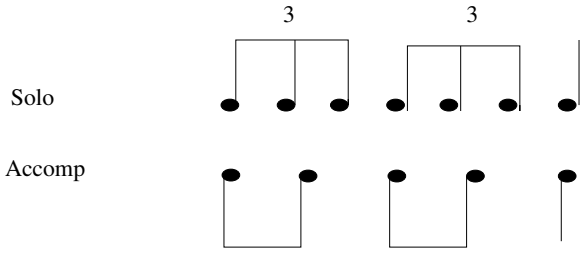
In the model above, l_k is the length of the kth note in *beats*, thus the product $l_k s_k$ is the time, in seconds, alloted to the kth note according to the local tempo, s_k. The variables σ_k and τ_k allow the model to be more flexible and to capture human aspects of interpretation. For instance, if $\sigma_k > 0$ this means that the tempo (in secs/beat) is increasing at the kth note, thus we are slowing down. Similarly, a negative value of σ_k denotes a local speeding up of tempo. The τ_k variables capture variations in note length that cannot be explained by tempo, such as in the case of a single note that is elongated without any decrease in tempo, as in an *agogic* accent.

More formally the pair σ_k, τ_k are modeled as Gaussian variables:

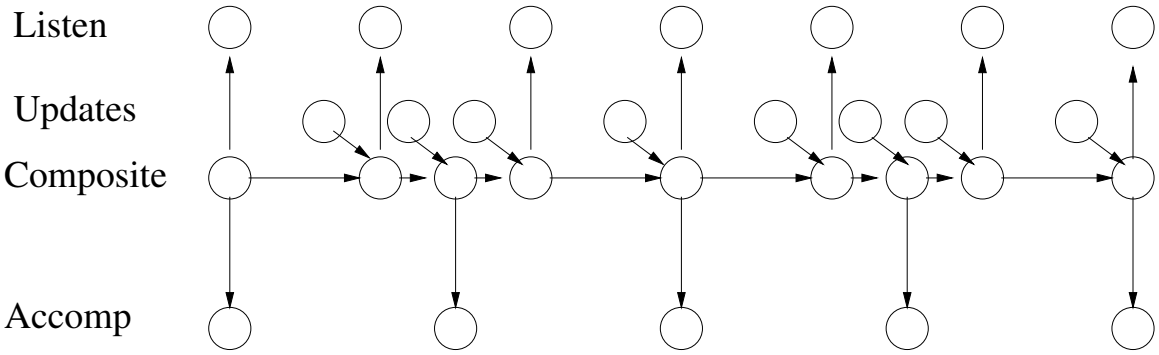
$$\begin{aligned} \tau_k &\sim N(\mu_{\tau_k}, \rho^2_{\tau_k}) \\ \sigma_k &\sim N(\mu_{\sigma_k}, \rho^2_{\sigma_k}) \end{aligned}$$

where $\{\tau_k, \sigma_k\}$ are a collection of independent random variables. In this notation $N(\mu, \rho^2)$ denotes a normal distribution with mean μ and variance ρ^2. With this modeling, μ_{τ_k} gives the average tempo change at a particular note while $\rho^2_{\tau_k}$ expressed the repeatability of this tempo change — is it nearly the same way every time (small $\rho^2_{\tau_k}$) or is it more variable from time to time (large $\rho^2_{\tau_k}$). Similar interpretations can be made of the mean and variance of the σ_k distribution.

The above model describe the time evolution of a single musical voice with a time varying tempo, "driven" by a collection of random variables. We now wish to treat the solo and accompaniment parts together. To this end, we will employ the model just discussed on the *composite* rhythm — the rhythm achieved by overlaying the musical parts. For instance, consider the two rhythms for solo and accompaniment described below:



This musical notation means that there are three evenly spaced notes in the solo part for every two evenly spaced notes in the accompaniment. In such a situation the two parts play some of the notes together (every 3rd note for the solo and every 2nd note for the accompaniment), while the others are not synchronous. This rhythm is expressed in our model by the following graph structure.

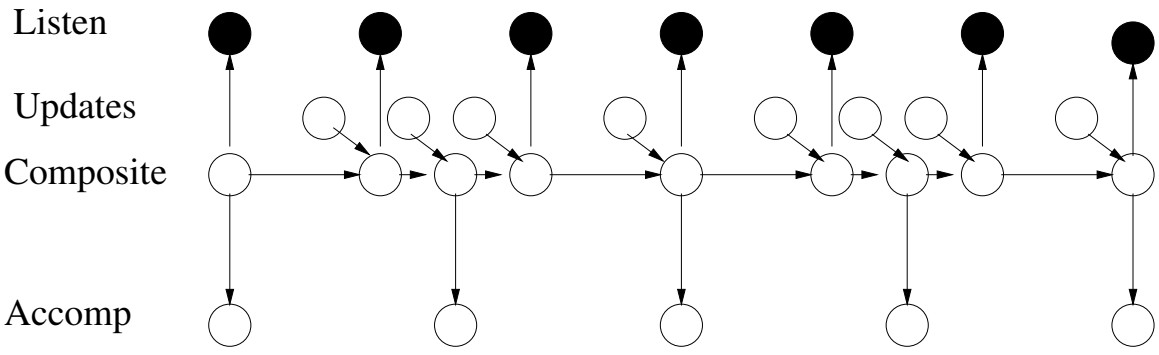


The expression of a probability distribution as a directed (with arrows) acyclic (no circuits) graph is a common way of expressing a probability distribution. The meaning of such a graph is essentially a *factorization* of the joint probability distribution on all variables. That is, the joint distribution is expressed as a product of conditional distributions, one for each graph node, in which the variables associated with the graph node depend only on their "parents" in the graph.

In interpreting this graph, the middle layer is the collection of time and tempo (t_k, s_k) variables for the composite rhythm describes by the model. The layer labeled as "update" are the random variables σ_k, τ_k that "drive" the model. The top layer of variables are the observed onset times estimated by the score following module while the lowest layer in the graph are the onset times of the accompaniment notes.

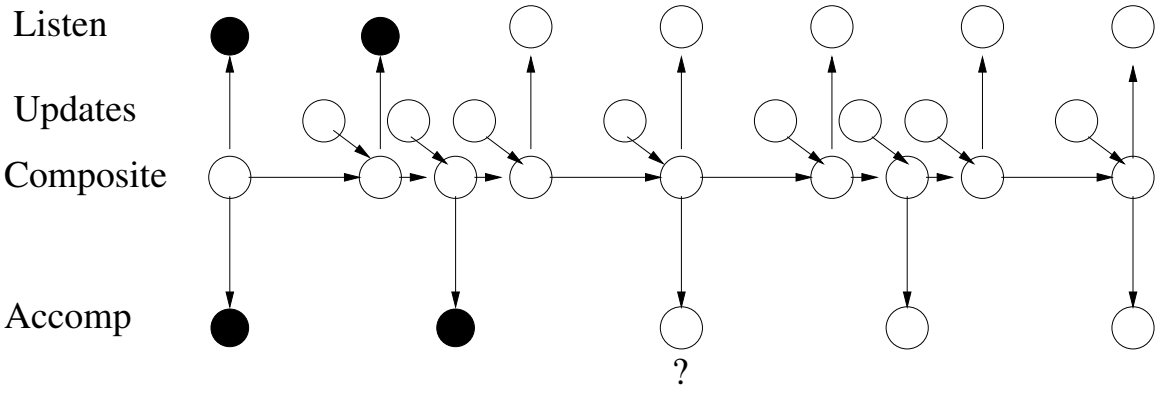
Our model facilitates a number of calculations that will be of interest to us, which we describe here. Each of these requires we compute the distribution on some of the unobserved variables of the model, given that we have observed other variables in the model. The mechanics of these computations will remain outside the scope of this course, though they are standard calculations for probabilistic graphical models and Bayesian belief networks.

The first problem is one problem we have already seen: given the complete solo performance, performed in isolation, how do we construct an orchestral accompaniment that fits to this. When discussed before we showed how phase vocoding could be used to warp and existing orchestral performance to fit the solo performance. Here we discuss the computation of the times at which the orchestra should play its notes. When we are given the soloist's times in our model we depict this by darkening in the circles corresponding to the observations, as below.



Given this information we can compute the conditional distribution on each of the variable corresponding to orchestra times. These times depend on the observed solo times, the raw score information, which is embedded in the model, and the nature of the "update" variables that capture the expression exhibited in past performances. We have already seen an application of this idea in our section of phase vocoding.

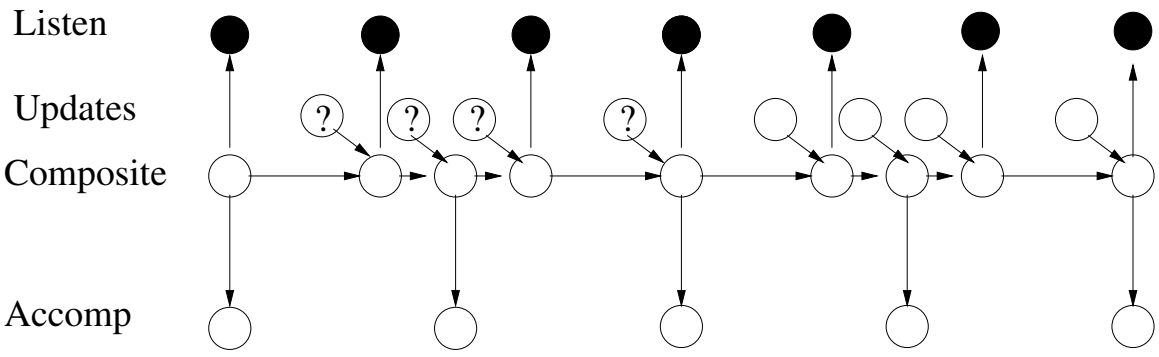
Another use of the model is for online musical accompaniment. In this scenario, the model is the backbone of the note scheduling procedure. For online accompaniment, as modeled here, the computer cares only about the scheduling of the *pending* accompaniment note. Using our model we can compute the conditional distribution of this time, given all of the currently-observed solo and accompaniment note. This information is depicted in the figure below.



Note that a particular accompaniment note may be scheduled and reschedule many times before actually being played. Each rescheduling will cause an adjustment of the playback rate so that the pending note is reached at the scheduled time.

A third use of the model is in *training* the accompaniment system to better follow a live player. In essence, this was the idea of the previous figure that shows the 5 performances of the same piece. In this example we saw considerable commonality *between* these performances, making one hope that a model that was trained to a specific player on a specific piece may be better able to anticipate the unfolding of future performances.

Recall that our musical interpretation is represented in the model in terms of the "update" $\{\tau_k, \sigma_k\}$ random variables which describe the way notes are stretched or compressed (the τ_k's) and the places where there are tempo changes (the σ_k's). In addition the distributions (means and variances) of these variables capture how repeatable these tendencies are. While we can never directly observe the update variables, (they really "exist" only in the context of our model), we can estimate them using the basic machinery of belief networks. That is, when conditioning on the observed solo times for a performance, we can compute the *conditional* distribution on any unobserved variable, such as the updates, as indicated in the figure below.



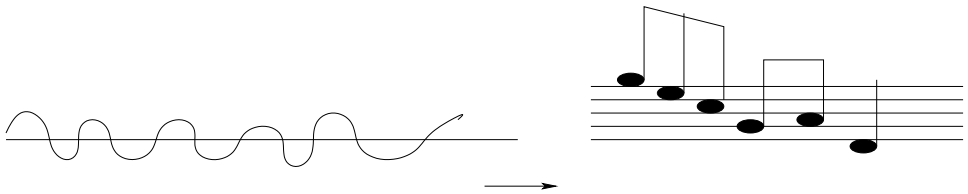
While we cannot observe the updates directly, we can compute their conditional mean times and use this as a substitute for direct knowledge. For instance, a natural estimate of the mean and variance of for one of these variables, say τ_k, would be to take the empirical mean and variance of the collection τ_k estimates (the conditional means) taken over a collection of performances. The estimation technique we will employ is a variation on this idea known as the EM algorithm, though we will not present the details here.

We will demonstrate in class the results of this kind of training technique in an actual performance situation.

New Music for Acoustic Instrument and Electronica

Music Transcription (Signal to Score)

The goal of music transcription is to transform an audio signal into a score-like symbolic representation as below:



In the most ambitions version of this problem we would like to recover the pitches, rhythms, instruments, voices, etc. for a piece of polyphonic music such as an orchestra, rock band, percussion ensemble, etc.

Possible applications of transcription include:

1. Create searchable, analyzable, printable symbolic database of any music genre. Searchability here may appeal to a music listener or librarian who wants to find a particular fragment of music — in essence, this seeks to be "Google for music." The analyzable aspect may appeal to a musicologist who wants to be more quantitative in the study of music than previous eras have allowed. The printable aspect will likely appeal to performs who would like to play the music. It is worth mentioning an argument against this approach, however. Optical character recognition for music is likely a much easier problem (though still quite challenging) This may be a faster way to assemble the desired symbolic databases for music that exists in printed form.

2. Capturing unnotated and improvisatory music, such as jazz or musics that are transmitted through an aural traditions.

3. This problem is a terrific intellectual challenge related to deep cognitive questions, (like how to model music).

4. Your answer here.

It is possible that the short term successes of transcription will be less important that those of score following, though there is much to offer here too.

Simply trying to pose the problem of transcription is a challenging task. The main difficulty here is in finding a simplification that allows practical solution, but not so simplified that the solutions are useless. In general this is an important aspect of much applied work. Here are some possible candidates:

1. Monophonic audio to score (as in "query by humming").

2. Audio to *piano roll* representation (discrete pitch and continuous start and stop times for notes).

3. Single instrument to score (piano, guitar, mbira, etc.)

4. Audio to chord sequence

5. rhythm transcription given onset times.

Of these the easiest is monophonic piano roll transcription in which we seek to partition an audio stream into segments labeled with a pitch or a rest. We will discuss this problem in a little more detail here.

To begin, recall the methods we discussed for the pitch recognition of a single frame. These were template matching, in which we seek the pitch or rest label t_k giving the best match to an audio frame s:

$$\hat{k} = \arg\max_k s \cdot t_k$$

or our maximum likelihood technique in which we sought the model t_k giving the highest probability to that data. This turned out to be the same as solving

$$\hat{k} = \arg\max_k s \cdot \log t_k$$

We saw that this technique performed okay for single frames but still made plenty of mistakes. The idea here is that now we seek a *sequence* of frame labels. How can we restrict our attention to sequences that make musical sense, or, at least, how can we *bias* our approach to favor musically plausible sequences.

For instance, suppose we run a frame recognizer and find it produces the sequence:

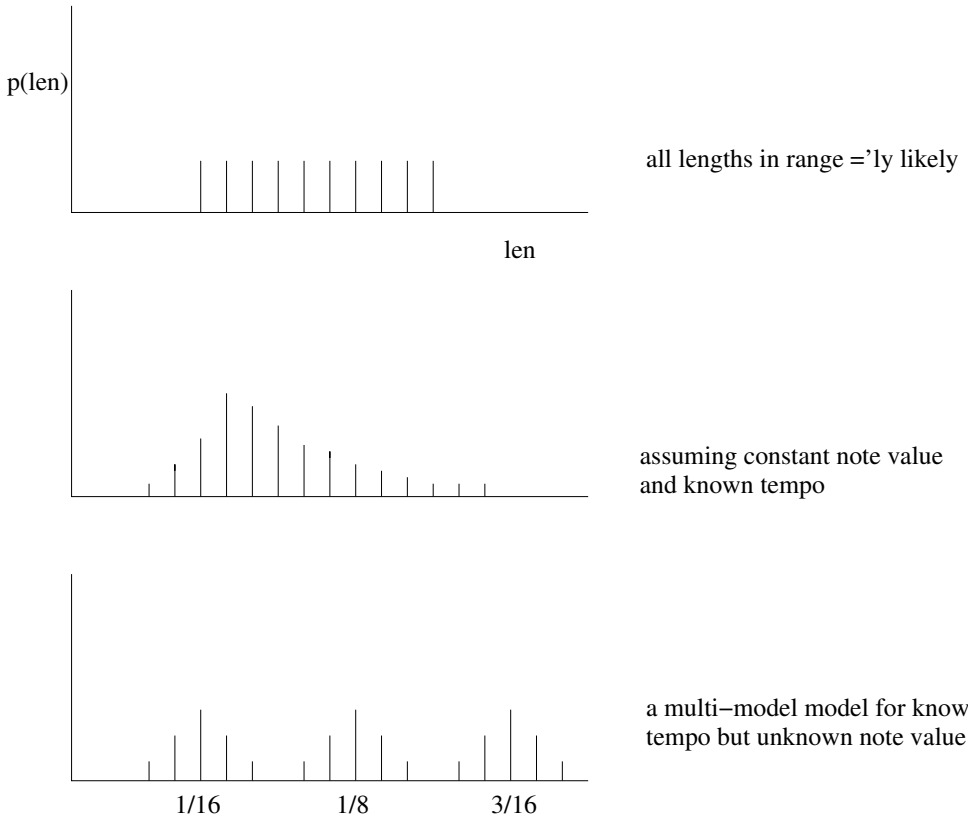
$$CC\sharp CCBCCCCCC\sharp DE\flat DDDC\sharp DDDD\ldots$$

It is, of course, impossible to have notes composed of single frames. A more likely interpretation would be

$$CCCCCCCCCCC|DDDDDDDDDD\ldots$$

HMM Approach (hmm_mono_recog.r)

Suppose we begin with a known note length probability distribution, in frames. Possible examples are given below where we assume a uniform distribution on note length, a more realistic distribution when both tempo and note value are known, and a third example that uses a multi-modal (several modes = peaks) trying to model the situation when we don't know the note value.



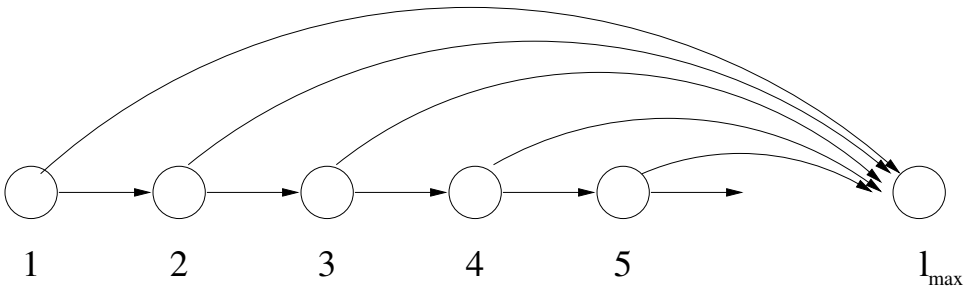
In all of these cases it is easy to build a model that captures the note length. If the note length distribution, in units of frames, is given by $q(l)$, with $q(l)$ possibly equal to 0 for some values and with maximal value $l_{\max}$, then we can construct the distribution through a collection of states $s_1, s_2, \ldots, s_{l_{\max}}$ where state s_l represents the situation in which the note is greater than l frames long. Thus we will only allow transitions from state s_l and states s_{l+1} and $s_{l_{\max}}$ with the transition probabilities

$$\begin{aligned} p(s_{l+1}|s_l) &= p(\text{len} \geq l+1 | \text{len} \geq l) \\ &= \frac{\sum_{k>l+1} q(k)}{\sum_{k>l} q(k)} \end{aligned}$$

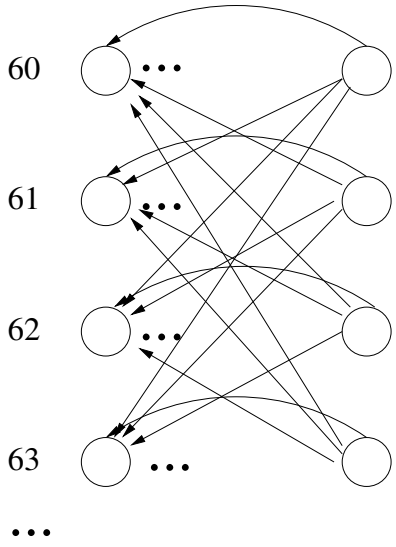
and

$$p(s_{l_{\max}}|s_l) = 1 - p(s_{l+1}|s_l)$$

The topology of this model is depicted below.



Once we have constructed such a note length model, we build a different instance of it for every possible pitch. Denoting these first and last states of these models in the diagrams below, we can form a "completely connected" graph that allows every possible pitch to follow every possible pitch as depicted below.



With the model in place, dynamic programming can be used to find the most likely path, given the data, using a data model such as the one already presented. A demonstration of this approach can be found in the program **hmm_mono_recog.r**.

Polyphonic Transcription

Polyphonic transcription tries to automatically represent music consisting of several independent parts. This problem seems to be more difficult than monophonic transcription by orders of magnitude.

For one, in monophonic transcription there are only a small number of possible note hypotheses, since a monophonic instrument only produces a small number of notes, say around 50. Furthermore, the audio signatures of many of these notes are quite dissimilar from one another, making certain distinctions easy to make. In polyphonic transcription, on the other hand, several notes may sound at once, leading to a large number of possible "chords" or "simultaneities." For example, if four instruments each play one of 50 notes, we have $50 \times 50 \times 50 \times 50 = 6,250,000$ possible configurations. To make matters worse, many of these are virtually indistinguishable using the kinds of techniques we have discussed. For instance, if both the notes C4 (middle C) and C5 sound at the same time, we only observe energy lying in the harmonic series of C4 — the C5 contribution is a subset of C4. There are many different combinations of pitches that lead to identical frequency signatures, sometimes known as "homonyms." This terms usually refers to two words that sound alike but are spelled differently, such as *aisle*, *I'll* and *isle*, but is also apt with pitch recognition. Perhaps, more to the point, many of these pitch configurations may not be identical, but are quite close to on another, making subtle distinctions quite challenging.

Due to the difficulty with polyphonic recognition, it is not surprising that a good deal of effort has focused on simpler problems that may be someday be pieces of a larger solution. Chord Identification is one example of this that has received a fair amount of attention in recent years. The idea here is simply to take a single audio spectrum, presumed to come from a single pitch configuration, and to identify the constituent pitches. A typical approach here would iteratively assemble a chord hypothesis in a "greedy" manner. That is, given an audio magnitude spectrum, s, we search for the single note spectrum t_k maximizing some measure of closeness $s \cdot \log t_k$. We can let

$$\hat{k}_1 = \arg\max_k s \cdot \log t_k$$

Then we proceed to try all other notes that can be added to $\hat{k}_1$. We can approximate the effect of two simultaneous notes simply by averaging their magnitude spectra: $(t_k + t_{k'})/2$. While the effect of two simultaneous notes is not really additive, it is difficult to come up with a more reasonable assumption due to the unknown nature of the interference between harmonics. Thus we could seek a 2nd pitch $\hat{k}_2$ by

$$\hat{k}_2 = \arg\max_k s \cdot \log(t_{\hat{k}_1} + t_k)/2$$

The general idea is to continue adding in more pitches until we can no longer improve the data score.

In practice, approaches like this may perform acceptably in some cases, but generally fall far short of "solving" the problem. In fact, there are no present approaches to polyphonic music recognition that seem all that promising if human abilities are the measure of success. There are several reasons why this may be the case.

One possibility is that we simply have not found the right way to represent the data. The spectrogram is an intuitively appealing representation, since, at first, it seems to capture so much of what we hear. Certainly one aspect of human hearing follows frequency content over time. However, the spectrogram fails to represent the phase information between frames that may be crucial to how we actually hear: musical sounds are so often objects that evolve continuously in amplitude, frequency, and phase. Perhaps a representation that decomposes the original sound into such continuously-varying building blocks may prove more useful for recognizing music. At any rate, the question of the basic data representation remains open.

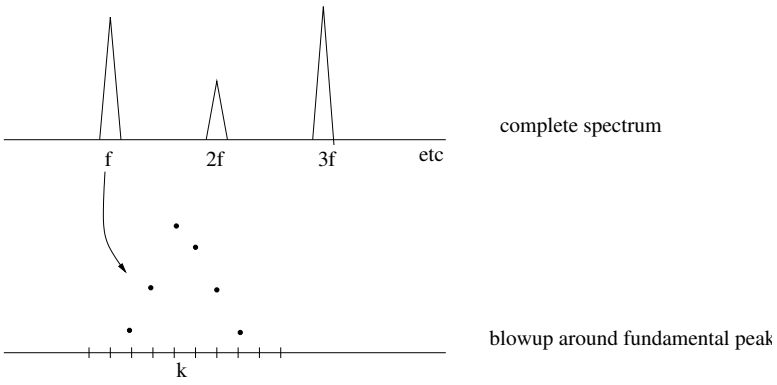
Another school of thought looks at the great amount of human knowledge we bring to music recognition. Much of this knowledge understands musical behaviors that "make sense" and are thus likely explanations. Repetition is a prime example of this kind of knowledge. A voice or musical part that moves back and forth between two pitches is likely to continue doing so. There are many generalizations of the notion of repetition, and all of these are integral to the kinds of expectations people have about how music will evolve. Other examples of human understanding and expectation happen on bigger time scales. For instance, much music is tonal, meaning that there is a certain most important note, and usually a corresponding "scale" of pitches. This creates strong beliefs in the listener about what notes makes sense in a tonal context. Perhaps music recognition that simultaneously recognizes such a tonal context

— say the key — may give better results. The same can be said about rhythmic considerations which also produce expectations on the listener's part. The simplest such example would be meter. Metered music decomposes music into units called measures where every measure contain the same amount musical time (beats). Typically there are patterns that appear at the same position in the measure as well. Simultaneously recognizing such high-level rhythmic notions may also improve our ability to transcribe music.

While simultaneously recognizing such musical structure may lead to better results, it also carries an added computation burden with it. While I personally believe this may well be the key to producing better recognition results, it would be fair to say there is not much evidence to back up this belief. Clearly, the polyphonic recognition problem remains open, and may continue so for a long time.

Precise Estimation of Pitch

As we have discussed, a Fourier transform of a pitched sound produces peaks around the harmonics. Both the lowest peak and the spacing between the peaks give the "fundamental" frequency, which we hear as the pitch of the sound. For instance, the figure below shows a typical magnitude spectrum for a pitched sound, with a "blowup" of the region around the fundamental peak.



From this picture we would be inclined to estimate the pitch of the sound by taking the "bin," $\hat{k}$, giving the maximum amplitude,

$$\hat{k} = \arg\max_k |X(k)|$$

where $X(k)$ is the Fourier transform, and estimating the frequency, in Hz, by

$$\hat{k}\ \frac{\text{cycles}}{\text{FFT length}} = \hat{k}\ \frac{\text{cycles}}{N/SR\ \text{secs}} = \frac{\hat{k} \cdot SR}{N}\ \frac{\text{cycles}}{\text{sec}}$$

Since $\hat{k}$ is an integer, our estimate is only accurate to SR/N Hz. For example, if we choose a sampling rate of 8 kHz with 1024 points per FFT, our estimate would be accurate to $\frac{8000}{1024} \approx 8$ Hz. What if we need a more accurate estimate?

Autocorrelation (autocor.r)

The autocorrelation looks at the "dot product" of the signal with shifted copies of itself. More precisely, the autocorrelation at "lag" r, for $r = 0, 1, \ldots$, computes the dot product of the signal and the signal shifted by r samples. That is, if the signal is $x_0, x_1, \ldots, x_{N-1}$ the autocorrelation is given by

$$a(r) = \frac{1}{N} \sum_n x_n x_{n+r}$$

for $r = 0, \ldots, N-1$. As with the discrete convolution discussed before, we take $n + r$ to mean the sum modulo N.

The autocorrelation is useful for detecting periodicities in the signal. To see this note that when a signal has period r samples, then $a(r)$ will be a dot product of the signal with itself. By the same reasoning employed for template matching, $a(r)$ will be the maximal value of the autocorrelation (along with $a(0), a(2r)$ etc.). Thus we find periodicities by looking for peaks of the autocorrelation.

To find the frequency in Hz suggested by the autocorrelation, a, suppose

$$\hat{r} = \arg\max_{r>0} a(r)$$

Then the frequency of the signal will be

$$\hat{r}\frac{\text{samples}}{\text{cycle}} = \frac{\hat{r}}{SR}\frac{\text{secs}}{\text{cycle}} = \frac{SR}{\hat{r}}\frac{\text{cycles}}{\text{sec}}$$

Thus the possible frequency estimates are SR/r for $r = 1, 2, 3, \ldots, N$. Notice that the autocorrelation does not produce uniform frequency resolution, with better resolution for low frequencies with longer period (large $\hat{r}$). However, both the autocorrelation and the Fourier method described above suffer from the discrete nature of the estimate, either in terms of frequency bins, or lag.

While we will not give an explanation, there is an interesting connection between the autocorrelation, $a(r)$ and the FFT X. That is, if we let

$$\begin{array}{ccc} a & \overset{\text{FFT}}{\longleftrightarrow} & A \\ x & \overset{\text{FFT}}{\longleftrightarrow} & X \end{array}$$

then we have $A(k) = X(k)\bar{X}(k) = |X(k)|^2$, where $\bar{X}$ denotes complex conjugate. The FFT of the autocorrelation is the squared modulus of the signal FFT.

Estimating Frequency using the Definition

Suppose we have a sampled sinusoid that oscillates at 1 cycle per FFT length. Imagine that we compute the STFT using a hop size of N/H where N is the FFT length. For each computed FFT, we will see a strong peak at bin 1; but what will happen with the phase at this peak? Since the signal oscillates at 1 cycle per N points, we will see the phase advancing by increments of $\frac{2\pi}{H}$ each FFT. From this one can see that the phase advance per hop is directly proportional to the frequency of the signal. Thus we could compute the frequency by computing the phase advance per hop, divided by the time associated with the hop. In other words, frequency is the rate of change or derivative of phase divided by 2π:

$$2\pi f(t) = \frac{d}{dt}\phi(t)$$

We can use this idea to estimate frequency directly from this definition. Suppose we observe a peak in our signal at bin k. We have already seen that a crude estimate of the frequency is $k\frac{SR}{N}$ Hz. To get a more precise estimate suppose that the next FFT is computed after "hopping" N samples. If the frequency were exactly k cycles per FFT length, we would see the same phase at the adjacent FFT for bin k. However, a more accurate estimate of the number of cycles traversed would be $k + (\phi_2(k) - \phi_1(k))/2\pi$ where $\phi_1(k)$ and $\phi_2(k)$ are the phases for bin k on the two transforms. This leads to the frequency estimate

$$\hat{f} = \frac{k + (\phi_2(k) - \phi_1(k))/2\pi}{N/SR}$$

In interpreting this formula, the difference $\phi_2(1) - \phi_1(1)$ is always taken to be a number in $(-\pi, \pi]$ — In other words, we assume that k is the best integer approximation to the number of cycles traversed between frames.

This same idea can be extended to the case with hop size N/H, and, in fact the estimation is usually more accurate as H increases. Suppose k is the bin containing a peak for two adjacent overlapping FFTs with hop N/H. If k were the exact frequency (in cycles per FFT length) we would expect the phase to advance by $2\pi k/H$ between the two frames. Thus, a more precise estimate of the phase traversed in this time window is $2\pi k + (\phi_2(k) - \phi_1(k)) - 2\pi k/H$. This leads to the frequency estimate

$$\hat{f} = \frac{2\pi k/H + ((\phi_2(k) - \phi_1(k)) - 2\pi k/H)_{-\pi}^{\pi}}{2\pi(N/H)/SR}$$

Here we denote the "principal argument" $(\Delta)_{-\pi}^{\pi}$ as the angle Δ viewed as lying in the range $(-\pi, \pi]$. That is

$$(\Delta)_{-\pi}^{\pi} = ((\Delta + \pi) \bmod 2\pi) - \pi$$